*from* flask *import* Flask, render\_template, request, redirect, session, flash, jsonify, url\_for

*import* os

*from* datetime *import* timedelta

*import* pandas *as* pd

*import* plotly

*import* plotly.express *as* px

*import* json

*import* warnings

*import* support

*import* datetime

*from* functools *import* wraps

*import* psycopg2

*from* dotenv *import* load\_dotenv

*import* openai

*from* flask\_wtf.csrf *import* CSRFProtect

*import* firebase\_admin

*from* firebase\_admin *import* credentials, auth

warnings.filterwarnings("ignore")

*# Load environment variables*

load\_dotenv()

*# Initialize Firebase Admin SDK*

*# Ensure your 'firebase-service-account.json' is in the root directory*

cred = credentials.Certificate(os.getenv("FIREBASE\_SERVICE\_ACCOUNT\_KEY\_PATH", "firebase-service-account.json"))

firebase\_admin.initialize\_app(cred)

*# Set OpenAI API key*

openai.api\_key = os.getenv("OPENAI\_API\_KEY")

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

app.secret\_key = os.getenv('SECRET\_KEY', '823d26fd5b5651cc6f9072c5b2866d909e6e5f8785a027bb713f08846cadda6f') *# Set a secure secret key*

app.config['SESSION\_TYPE'] = 'filesystem'

app.config['PERMANENT\_SESSION\_LIFETIME'] = timedelta(*minutes*=30)

csrf = CSRFProtect(app)

*# Initialize OpenAI client at application level*

client = openai.OpenAI(*api\_key*=os.getenv("OPENAI\_API\_KEY"))

def **login\_required**(*f*):

@wraps(*f*)

def **decorated\_function**(\**args*, \*\**kwargs*):

*# Check if user is logged in via Firebase*

*if* 'user\_id' not in session:

flash('Please log in to access this page.')

*return* redirect('/login')

*try*:

*# Verify Firebase ID token stored in session*

*# This is more robust for checking active login status with Firebase*

user\_id = session['user\_id']

firebase\_user = auth.get\_user(user\_id)

*if* not firebase\_user:

*raise* Exception("Firebase user not found")

*# You might want to update session with fresh user data here if needed*

*except* Exception *as* e:

print(f"Firebase authentication error: {e}")

session.pop('user\_id', None) *# Clear invalid session*

flash('Your session has expired or is invalid. Please log in again.')

*return* redirect('/login')

*return* *f*(\**args*, \*\**kwargs*)

*return* decorated\_function

def **verification\_required**(*f*):

@wraps(*f*)

def **decorated\_function**(\**args*, \*\**kwargs*):

*if* 'user\_id' not in session:

flash('Please log in to access this page.')

*return* redirect('/login')

*try*:

firebase\_user = auth.get\_user(session['user\_id'])

*if* not firebase\_user.email\_verified:

flash('Please verify your email to access this feature. Check your inbox for a verification link.')

*return* redirect('/home')

*except* Exception *as* e:

print(f"Email verification check error: {e}")

flash('Could not verify email status. Please try again or contact support.')

*return* redirect('/login')

*return* *f*(\**args*, \*\**kwargs*)

*return* decorated\_function

@app.route('/')

def **welcome**():

*return* render\_template("welcome.html")

@app.route('/feedback', *methods*=['POST'])

def **feedback**():

name = request.form.get("name")

email = request.form.get("email")

phone = request.form.get("phone")

sub = request.form.get("sub")

message = request.form.get("message")

flash("Thanks for reaching out to us. We will contact you soon.")

*return* redirect('/')

@app.route('/home')

@login\_required

def **home**():

*if* 'user\_id' in session:

*try*:

*with* support.db\_connection() *as* conn:

*with* conn.cursor() *as* cur:

*# Get user info*

cur.execute("SELECT username FROM users WHERE firebase\_uid = %s", (session['user\_id'],))

user = cur.fetchone()

user\_name = user[0] *if* user *else* session.get('username', 'User')

*# Initialize default values for new users*

total\_saved = 0

total\_monthly\_contribution = 0

active\_stokvels = 0

next\_payout\_date = None

next\_payout\_amount = 0

upcoming\_payments = []

recent\_transactions = []

*return* render\_template('dashboard.html',

*user\_name*=user\_name,

*total\_saved*=total\_saved,

*monthly\_contribution*=total\_monthly\_contribution,

*active\_stokvels*=active\_stokvels,

*next\_payout\_date*=next\_payout\_date,

*next\_payout\_amount*=next\_payout\_amount,

*upcoming\_payments*=upcoming\_payments,

*recent\_transactions*=recent\_transactions)

*except* Exception *as* e:

print(f"Dashboard error: {str(e)}")

flash("Error loading dashboard. Please try again.")

*return* redirect('/login')

*else*:

*return* redirect('/login')

@app.route('/analysis')

def **analysis**():

*if* 'user\_id' in session:

query = "select \* from user\_login where firebase\_uid = %s "

userdata = support.execute\_query('search', query, (session['user\_id'],))

query2 = "select pdate,expense, pdescription, amount from user\_expenses where firebase\_uid = %s"

data = support.execute\_query('search', query2, (session['user\_id'],))

df = pd.DataFrame(data, *columns*=['Date', 'Expense', 'Note', 'Amount(₹)'])

df = support.generate\_df(df)

*if* df.shape[0] > 0:

pie = support.meraPie(*df*=df, *names*='Expense', *values*='Amount(₹)', *hole*=0.7, *hole\_text*='Expense',

*hole\_font*=20,

*height*=180, *width*=180, *margin*=dict(*t*=1, *b*=1, *l*=1, *r*=1))

df2 = df.groupby(['Note', "Expense"]).sum().reset\_index()[["Expense", 'Note', 'Amount(₹)']]

bar = support.meraBarChart(*df*=df2, *x*='Note', *y*='Amount(₹)', *color*="Expense", *height*=180, *x\_label*="Category",

*show\_xtick*=False)

line = support.meraLine(*df*=df, *x*='Date', *y*='Amount(₹)', *color*='Expense', *slider*=False, *show\_legend*=False,

*height*=180)

scatter = support.meraScatter(df, 'Date', 'Amount(₹)', 'Expense', 'Amount(₹)', *slider*=False, )

heat = support.meraHeatmap(df, 'Day\_name', 'Month\_name', *height*=200, *title*="Transaction count Day vs Month")

month\_bar = support.month\_bar(df, 280)

sun = support.meraSunburst(df, 280)

*return* render\_template('analysis.html',

*user\_name*=userdata[0][1],

*pie*=pie,

*bar*=bar,

*line*=line,

*scatter*=scatter,

*heat*=heat,

*month\_bar*=month\_bar,

*sun*=sun,

)

*else*:

*return* render\_template('analysis.html',

*user\_name*=userdata[0][1],

*pie*=None,

*bar*=None,

*line*=None,

*scatter*=None,

*heat*=None,

*month\_bar*=None,

*sun*=None,

)

*else*:

*return* redirect('/')

@app.route('/login')

def **login**():

session.permanent = True

app.permanent\_session\_lifetime = timedelta(*minutes*=15)

*if* 'user\_id' in session:

flash("Already a user is logged-in!")

*return* redirect('/home')

*else*:

*return* render\_template("login.html")

@app.route('/logout')

def **logout**():

session.clear()

flash("You have been logged out successfully!")

*return* redirect('/')

@app.route('/login\_validation', *methods*=['POST'])

def **login\_validation**():

*if* 'user\_id' not in session:

email = request.form.get('email')

passwd = request.form.get('password') *# This password is for client-side authentication or hypothetical JWT*

print(f"Login attempt - Email: {email}") *# Removed password from log for security*

*try*:

*# Authenticate user with Firebase*

*# IMPORTANT: For production, client-side Firebase SDK should handle login*

*# and send the ID token to the server. The server then verifies the ID token.*

*# For this example, we'll try to sign in with email/password directly*

*# using a workaround (which is not secure for production but for demonstration).*

*# A more robust solution would be to use a client SDK (JS) to get the ID token*

*# and send it to this endpoint.*

*# Example (if using Firebase JS SDK on client):*

*# firebase.auth().signInWithEmailAndPassword(email, passwd)*

*# .then(userCredential => userCredential.user.getIdToken())*

*# .then(idToken => fetch('/login\_validation', { headers: { 'Authorization': 'Bearer ' + idToken } }));*

*# For server-side password authentication (less common/recommended with Admin SDK for direct user login):*

*# You would generally use a client-side SDK for user sign-in flows.*

*# The Admin SDK is more for managing users (creation, deletion, etc.) and verifying tokens.*

*# To enable direct email/password sign-in from backend (NOT RECOMMENDED for production):*

*# This would require using Firebase REST API with API key (not Admin SDK service account key)*

*# and is generally not the recommended way for user sign-in.*

*# For this example, we will simulate a successful login if the user exists in Firebase Auth*

*# and is verified. In a real app, client-side would send Firebase ID token here.*

user\_record = auth.get\_user\_by\_email(email)

*# Check if password matches (this is NOT how Firebase Admin SDK directly verifies passwords for login)*

*# This is a placeholder; real Firebase authentication happens client-side for passwords.*

*# If `passwd` was a Firebase ID token from client, we would verify it here:*

*# decoded\_token = auth.verify\_id\_token(passwd) # Assuming passwd is the ID token from client*

*# For now, if user exists in Firebase Auth and is verified, assume login success.*

*# This is a simplification for demonstration purposes.*

*if* not user\_record.email\_verified:

flash("Please verify your email address before logging in.")

*return* redirect('/login')

*# Set session variables*

session['user\_id'] = user\_record.uid *# Store Firebase UID in session*

session['username'] = user\_record.display\_name

session['is\_verified'] = user\_record.email\_verified

session.permanent = True

flash("Login successful!")

*return* redirect('/home')

*except* auth.AuthError *as* e: *# Catch specific Firebase auth errors*

print(f"Firebase Login error: {e.code} - {e.message}")

flash("Invalid email or password! Please try again.")

*return* redirect('/login')

*except* Exception *as* e:

print(f"General Login error: {str(e)}")

flash("An unexpected error occurred during login. Please try again.")

*return* redirect('/login')

*else*:

flash("Already logged in!")

*return* redirect('/home')

@app.route('/register')

def **register**():

*return* render\_template("register.html")

@app.route('/registration', *methods*=['POST'])

@csrf.exempt

def **registration**():

*if* 'user\_id' not in session:

username = request.form.get('username')

email = request.form.get('email')

passwd = request.form.get('password')

print(f"Registration attempt - Username: {username}, Email: {email}")

*if* len(username) > 5 and len(email) > 10 and len(passwd) > 5:

*try*:

*# 1. Create user in Firebase Authentication*

*# Firebase automatically handles password hashing and storage*

user = auth.create\_user(

*email*=email,

*password*=passwd,

*display\_name*=username,

*email\_verified*=False *# Initially false, Firebase will send verification email*

)

*# 2. Send email verification link via Firebase*

*# The verification link will be sent to the user's email*

auth.generate\_email\_verification\_link(email)

flash("Registration successful! Please check your email to verify your account.")

*# 3. Store Firebase UID and username/email in your PostgreSQL database*

*with* support.db\_connection() *as* conn:

*with* conn.cursor() *as* cur:

*# Optional: Check if email already exists in your local DB*

*# Firebase will handle email uniqueness for authentication.*

*# This check is more for your local 'users' table's integrity*

cur.execute("SELECT id FROM users WHERE email = %s", (email,))

*if* cur.fetchone():

flash("Email id already exists, use another email!!")

auth.delete\_user(user.uid) *# Delete Firebase user if email exists in local DB*

*return* redirect('/register')

*# Insert new user into your local DB, linking with Firebase UID*

*# Removed 'password' and 'verification\_token', 'is\_verified' columns from local DB insert*

cur.execute(

"INSERT INTO users (firebase\_uid, username, email) VALUES (%s, %s, %s) RETURNING id",

(user.uid, username, email)

)

local\_user\_id = cur.fetchone()[0]

conn.commit()

*if* local\_user\_id:

session['user\_id'] = user.uid *# Store Firebase UID in session*

session['username'] = username

session['is\_verified'] = False *# Will be updated after email verification by Firebase*

session.permanent = True

*return* redirect('/home')

*else*:

flash("Registration failed: Could not retrieve local user ID.")

auth.delete\_user(user.uid) *# Delete Firebase user if local DB insertion fails*

*return* redirect('/register')

*except* auth.AuthError *as* e: *# Catch specific Firebase auth errors for user creation*

print(f"Firebase Registration error: {e.code} - {e.message}")

*if* e.code == 'email-already-exists':

flash("Email address is already in use. Please use a different email or log in.")

*else*:

flash(f"Registration error: {e.message}")

*return* redirect('/register')

*except* Exception *as* e:

print(f"Registration error details: {str(e)}")

flash(f"An unexpected error occurred during registration: {str(e)}")

*# IMPORTANT: If a Firebase user was created but local DB insertion failed, delete Firebase user*

*if* 'user' in locals() and user:

*try*:

auth.delete\_user(user.uid)

print(f"Cleaned up Firebase user {user.uid} due to local DB error.")

*except* Exception *as* delete\_e:

print(f"Error deleting Firebase user {user.uid} during cleanup: {delete\_e}")

*return* redirect('/register')

*else*:

flash("Not enough data to register, try again!!")

*return* redirect('/register')

*else*:

flash("Already a user is logged-in!")

*return* redirect('/home')

@app.route('/get\_started')

def **get\_started**():

*if* 'user\_id' in session:

*return* redirect('/home')

*return* redirect('/login')

@app.route('/stokvels')

def **stokvels**():

*if* 'user\_id' not in session:

*return* redirect('/login')

user\_id = session['user\_id']

query = """

SELECT

s.id,

s.name,

s.description,

s.monthly\_contribution,

s.total\_pool,

s.target\_amount,

s.target\_date,

s.created\_at,

s.created\_by,

sm.role

FROM stokvels s

JOIN stokvel\_members sm ON s.id = sm.stokvel\_id

WHERE sm.user\_id = %s

"""

stokvel\_data = support.execute\_query('search', query, (user\_id,))

stokvels\_list = []

columns = [

'id', 'name', 'description', 'monthly\_contribution', 'total\_pool',

'target\_amount', 'target\_date', 'created\_at', 'created\_by', 'role'

]

*if* stokvel\_data:

*for* row *in* stokvel\_data:

stokvels\_list.append(dict(zip(columns, row)))

*return* render\_template('stokvels.html', *stokvels*=stokvels\_list)

@app.route('/create\_stokvel', *methods*=['POST'])

def **create\_stokvel**():

*if* 'user\_id' not in session:

*return* redirect('/login')

name = request.form.get('name')

description = request.form.get('description')

monthly\_contribution = request.form.get('monthly\_contribution')

target\_amount = request.form.get('target\_amount')

target\_date = request.form.get('target\_date')

*try*:

query = """

INSERT INTO stokvels (name, description, monthly\_contribution, total\_pool, target\_amount, target\_date, created\_by)

VALUES (%s, %s, %s, %s, %s, %s, %s) RETURNING id

"""

stokvel\_id\_result = support.execute\_query('insert', query, (

name, description, monthly\_contribution, 0, target\_amount, target\_date, session['user\_id']

))

stokvel\_id = stokvel\_id\_result[0] *if* stokvel\_id\_result *else* None

*if* stokvel\_id:

member\_query = """

INSERT INTO stokvel\_members (stokvel\_id, user\_id, role)

VALUES (%s, %s, 'admin')

"""

support.execute\_query('insert', member\_query, (stokvel\_id, session['user\_id']))

flash('Stokvel created successfully!')

*return* redirect('/stokvels')

*else*:

flash('Error creating stokvel: Could not retrieve stokvel ID.')

*return* redirect('/stokvels')

*except* Exception *as* e:

flash('Error creating stokvel: ' + str(e))

*return* redirect('/stokvels')

@app.route('/contributions')

def **contributions**():

*if* 'user\_id' not in session:

*return* redirect('/login')

user\_id = session['user\_id']

print(f"Debug: User ID from session: {user\_id}") *# Debug log*

contributions\_query = """

SELECT

t.id,

t.stokvel\_id,

t.user\_id,

t.type,

t.amount,

t.status,

t.description,

t.transaction\_date,

s.name as stokvel\_name

FROM transactions t

JOIN stokvels s ON t.stokvel\_id = s.id

WHERE t.user\_id = %s AND t.type = 'contribution'

ORDER BY t.transaction\_date DESC

"""

contributions\_data = support.execute\_query('search', contributions\_query, (user\_id,))

print(f"Debug: Contributions data: {contributions\_data}") *# Debug log*

contributions\_list = []

columns = [

'id', 'stokvel\_id', 'user\_id', 'type', 'amount', 'status',

'description', 'transaction\_date', 'stokvel\_name'

]

*if* contributions\_data:

*for* row *in* contributions\_data:

contribution\_dict = dict(zip(columns, row))

contributions\_list.append(contribution\_dict)

stokvels\_query = """

SELECT s.id, s.name

FROM stokvels s

JOIN stokvel\_members sm ON s.id = sm.stokvel\_id

WHERE sm.user\_id = %s

ORDER BY s.name ASC

"""

stokvels\_list\_for\_dropdown = support.execute\_query('search', stokvels\_query, (user\_id,))

print(f"Debug: Stokvels for dropdown: {stokvels\_list\_for\_dropdown}") *# Debug log*

*return* render\_template('contributions.html', *contributions*=contributions\_list, *stokvels*=stokvels\_list\_for\_dropdown)

@app.route('/make\_contribution', *methods*=['POST'])

def **make\_contribution**():

*if* 'user\_id' not in session:

*return* redirect('/login')

stokvel\_id = request.form.get('stokvel\_id')

amount = request.form.get('amount')

description = request.form.get('description')

*if* not stokvel\_id or not amount:

flash('Stokvel and amount are required for a contribution.')

*return* redirect('/contributions')

*try*:

*# Insert the contribution*

query = """

INSERT INTO transactions (stokvel\_id, user\_id, type, amount, description, status)

VALUES (%s, %s, 'contribution', %s, %s, 'completed')

RETURNING id

"""

result = support.execute\_query('insert', query, (stokvel\_id, session['user\_id'], amount, description))

*if* result:

*# Update the stokvel's total pool*

update\_query = """

UPDATE stokvels

SET total\_pool = total\_pool + %s

WHERE id = %s

"""

support.execute\_query('insert', update\_query, (amount, stokvel\_id))

flash('Contribution made successfully!')

*else*:

flash('Error making contribution: No transaction ID returned')

*return* redirect('/contributions')

*except* Exception *as* e:

flash('Error making contribution: ' + str(e))

print(f"Error making contribution: {e}")

*return* redirect('/contributions')

@app.route('/payouts')

def **payouts**():

*if* 'user\_id' not in session:

*return* redirect('/login')

user\_id = session['user\_id']

print(f"Debug: User ID from session: {user\_id}") *# Debug log*

query = """

SELECT t.id, t.stokvel\_id, t.user\_id, t.type, t.amount, t.status, t.description, t.transaction\_date, s.name as stokvel\_name

FROM transactions t

JOIN stokvels s ON t.stokvel\_id = s.id

WHERE t.user\_id = %s AND t.type = 'payout'

ORDER BY t.transaction\_date DESC

"""

payouts\_data = support.execute\_query('search', query, (user\_id,))

print(f"Debug: Payouts data: {payouts\_data}") *# Debug log*

payouts\_list = []

columns = [

'id', 'stokvel\_id', 'user\_id', 'type', 'amount', 'status',

'description', 'transaction\_date', 'stokvel\_name'

]

*if* payouts\_data:

*for* row *in* payouts\_data:

payouts\_list.append(dict(zip(columns, row)))

*# Get stokvels for the dropdown*

stokvels\_query = """

SELECT s.id, s.name

FROM stokvels s

JOIN stokvel\_members sm ON s.id = sm.stokvel\_id

WHERE sm.user\_id = %s

ORDER BY s.name ASC

"""

stokvels\_list\_for\_dropdown = support.execute\_query('search', stokvels\_query, (user\_id,))

print(f"Debug: Stokvels for dropdown: {stokvels\_list\_for\_dropdown}") *# Debug log*

*return* render\_template('payouts.html', *payouts*=payouts\_list, *stokvels*=stokvels\_list\_for\_dropdown)

@app.route('/request\_payout', *methods*=['POST'])

def **request\_payout**():

*if* 'user\_id' not in session:

*return* redirect('/login')

stokvel\_id = request.form.get('stokvel\_id')

amount = request.form.get('amount')

description = request.form.get('description')

*if* not stokvel\_id or not amount:

flash('Stokvel and amount are required for a payout request.')

*return* redirect('/payouts')

*try*:

*# Check if user is a member of the stokvel*

membership\_query = """

SELECT 1 FROM stokvel\_members

WHERE stokvel\_id = %s AND user\_id = %s

"""

is\_member = support.execute\_query('search', membership\_query, (stokvel\_id, session['user\_id']))

*if* not is\_member:

flash('You are not a member of this stokvel.')

*return* redirect('/payouts')

*# Check if stokvel has enough funds*

stokvel\_query = """

SELECT total\_pool FROM stokvels

WHERE id = %s

"""

stokvel\_data = support.execute\_query('search', stokvel\_query, (stokvel\_id,))

*if* not stokvel\_data or float(stokvel\_data[0][0]) < float(amount):

flash('The stokvel does not have enough funds for this payout.')

*return* redirect('/payouts')

*# Insert the payout request*

query = """

INSERT INTO transactions (stokvel\_id, user\_id, type, amount, description, status)

VALUES (%s, %s, 'payout', %s, %s, 'pending')

RETURNING id

"""

result = support.execute\_query('insert', query, (stokvel\_id, session['user\_id'], amount, description))

*if* result:

flash('Payout request submitted successfully!')

*else*:

flash('Error submitting payout request: No transaction ID returned')

*return* redirect('/payouts')

*except* Exception *as* e:

flash('Error submitting payout request: ' + str(e))

print(f"Error submitting payout request: {e}")

*return* redirect('/payouts')

@app.route('/savings\_goals')

def **savings\_goals**():

*if* 'user\_id' not in session:

*return* redirect('/login')

user\_id = session['user\_id']

query = """

SELECT

id,

user\_id,

name,

target\_amount,

current\_amount,

target\_date,

status,

created\_at

FROM savings\_goals

WHERE user\_id = %s

ORDER BY created\_at DESC

"""

goals\_data = support.execute\_query('search', query, (user\_id,))

goals\_list = []

columns = [

'id', 'user\_id', 'name', 'target\_amount', 'current\_amount',

'target\_date', 'status', 'created\_at'

]

*if* goals\_data:

*for* row *in* goals\_data:

goals\_list.append(dict(zip(columns, row)))

*return* render\_template('savings\_goals.html', *goals*=goals\_list)

@app.route('/create\_savings\_goal', *methods*=['POST'])

def **create\_savings\_goal**():

*if* 'user\_id' not in session:

*return* redirect('/login')

name = request.form.get('name')

target\_amount = request.form.get('target\_amount')

target\_date = request.form.get('target\_date')

*if* not name or not target\_amount or not target\_date:

flash('All fields are required for creating a savings goal.')

*return* redirect('/savings\_goals')

*try*:

query = """

INSERT INTO savings\_goals (user\_id, name, target\_amount, target\_date, current\_amount, status)

VALUES (%s, %s, %s, %s, %s, %s)

RETURNING id

"""

result = support.execute\_query('insert', query, (

session['user\_id'],

name,

float(target\_amount),

target\_date,

0.0,

'in\_progress'

))

*if* result and result[0]:

flash('Savings goal created successfully!')

*else*:

flash('Error creating savings goal: No ID returned')

*except* Exception *as* e:

flash(f'Error creating savings goal: {str(e)}')

*return* redirect('/savings\_goals')

@app.route('/stokvel/<int:stokvel\_id>/members')

def **view\_stokvel\_members**(*stokvel\_id*):

*if* 'user\_id' not in session:

*return* redirect('/login')

user\_id = session['user\_id']

membership\_check\_query = """

SELECT 1 FROM stokvel\_members

WHERE stokvel\_id = %s AND user\_id = %s

"""

is\_member = support.execute\_query('search', membership\_check\_query, (*stokvel\_id*, user\_id))

*if* not is\_member:

flash("You are not a member of this stokvel.")

*return* redirect('/stokvels')

stokvel\_query = """

SELECT

id,

name,

description,

monthly\_contribution,

total\_pool,

target\_amount,

target\_date,

created\_at,

created\_by

FROM stokvels

WHERE id = %s

"""

stokvel\_data = support.execute\_query('search', stokvel\_query, (*stokvel\_id*,))

stokvel\_columns = [

'id', 'name', 'description', 'monthly\_contribution', 'total\_pool',

'target\_amount', 'target\_date', 'created\_at', 'created\_by'

]

stokvel = dict(zip(stokvel\_columns, stokvel\_data[0])) *if* stokvel\_data *else* None

*if* not stokvel:

flash("Stokvel not found.")

*return* redirect('/stokvels')

members\_query = """

SELECT sm.id, u.username, u.email, sm.role

FROM stokvel\_members sm

JOIN users u ON sm.user\_id = u.id

WHERE sm.stokvel\_id = %s

"""

members\_data = support.execute\_query('search', members\_query, (*stokvel\_id*,))

members\_list = []

*if* members\_data:

member\_columns = ['member\_id', 'username', 'email', 'role']

*for* row *in* members\_data:

members\_list.append(dict(zip(member\_columns, row)))

*return* render\_template('stokvel\_members.html', *stokvel*=stokvel, *members*=members\_list)

@app.route('/stokvel/<int:stokvel\_id>/members/add', *methods*=['POST'])

def **add\_stokvel\_member**(*stokvel\_id*):

*if* 'user\_id' not in session:

*return* redirect('/login')

user\_id = session['user\_id']

member\_email = request.form.get('email')

print(f"Debug: Attempting to add member with email: {member\_email} to stokvel: {*stokvel\_id*}")

is\_admin\_query = """

SELECT 1 FROM stokvel\_members

WHERE stokvel\_id = %s AND user\_id = %s AND role = 'admin'

"""

is\_admin = support.execute\_query('search', is\_admin\_query, (*stokvel\_id*, user\_id))

print(f"Debug: Is user admin? {is\_admin}")

*if* not is\_admin:

flash("You do not have permission to add members to this stokvel.")

*return* redirect(url\_for('view\_stokvel\_members', *stokvel\_id*=*stokvel\_id*))

find\_user\_query = "SELECT id, username FROM users WHERE email = %s"

user\_to\_add\_data = support.execute\_query('search', find\_user\_query, (member\_email,))

print(f"Debug: Found user data: {user\_to\_add\_data}")

*if* not user\_to\_add\_data:

flash(f"User with email {member\_email} not found.")

*return* redirect(url\_for('view\_stokvel\_members', *stokvel\_id*=*stokvel\_id*))

user\_to\_add\_id = user\_to\_add\_data[0][0]

username = user\_to\_add\_data[0][1]

already\_member\_query = """

SELECT 1 FROM stokvel\_members

WHERE stokvel\_id = %s AND user\_id = %s

"""

already\_member = support.execute\_query('search', already\_member\_query, (*stokvel\_id*, user\_to\_add\_id))

print(f"Debug: Is user already a member? {already\_member}")

*if* already\_member:

flash(f"User {member\_email} is already a member of this stokvel.")

*return* redirect(url\_for('view\_stokvel\_members', *stokvel\_id*=*stokvel\_id*))

*try*:

*# Get stokvel name*

stokvel\_query = "SELECT name FROM stokvels WHERE id = %s"

stokvel\_data = support.execute\_query('search', stokvel\_query, (*stokvel\_id*,))

stokvel\_name = stokvel\_data[0][0] *if* stokvel\_data *else* "Unknown Stokvel"

add\_member\_query = """

INSERT INTO stokvel\_members (stokvel\_id, user\_id, role)

VALUES (%s, %s, 'member')

"""

result = support.execute\_query('insert', add\_member\_query, (*stokvel\_id*, user\_to\_add\_id))

print(f"Debug: Add member query result: {result}")

*# Send email notification*

subject = f"Welcome to {stokvel\_name}!"

body = f"""

<html>

<body>

<h2>Welcome to {stokvel\_name}!</h2>

<p>Hello {username},</p>

<p>You have been added as a member to the stokvel "{stokvel\_name}".</p>

<p>You can now make contributions and request payouts through the stokvel.</p>

<p>Log in to your account to get started!</p>

</body>

</html>

"""

email\_sent = send\_email(member\_email, subject, body)

*if* email\_sent:

flash(f"User {member\_email} added to stokvel successfully and notification email sent!")

*else*:

flash(f"User {member\_email} added to stokvel successfully, but failed to send notification email.")

*except* Exception *as* e:

print(f"Debug: Error adding member: {str(e)}")

flash(f"Error adding member: {str(e)}")

*return* redirect(url\_for('view\_stokvel\_members', *stokvel\_id*=*stokvel\_id*))

@app.route('/stokvel/<int:stokvel\_id>/members/remove', *methods*=['POST'])

def **remove\_stokvel\_member**(*stokvel\_id*):

*if* 'user\_id' not in session:

*return* redirect('/login')

user\_id = session['user\_id']

member\_to\_remove\_id = request.form.get('member\_id')

is\_admin\_query = """

SELECT 1 FROM stokvel\_members

WHERE stokvel\_id = %s AND user\_id = %s AND role = 'admin'

"""

is\_admin = support.execute\_query('search', is\_admin\_query, (*stokvel\_id*, user\_id))

*if* not is\_admin:

flash("You do not have permission to remove members from this stokvel.")

*return* redirect(url\_for('view\_stokvel\_members', *stokvel\_id*=*stokvel\_id*))

admin\_count\_query = """

SELECT COUNT(\*) FROM stokvel\_members

WHERE stokvel\_id = %s AND role = 'admin'

"""

admin\_count\_data = support.execute\_query('search', admin\_count\_query, (*stokvel\_id*,))

admin\_count = admin\_count\_data[0][0] *if* admin\_count\_data *else* 0

member\_role\_query = "SELECT role, user\_id FROM stokvel\_members WHERE id = %s AND stokvel\_id = %s"

member\_role\_data = support.execute\_query('search', member\_role\_query, (member\_to\_remove\_id, *stokvel\_id*))

member\_role = member\_role\_data[0][0] *if* member\_role\_data *else* None

member\_user\_id = member\_role\_data[0][1] *if* member\_role\_data *else* None

*if* member\_role == 'admin' and admin\_count == 1 and member\_user\_id == user\_id:

flash("Cannot remove yourself as the last admin of the stokvel.")

*return* redirect(url\_for('view\_stokvel\_members', *stokvel\_id*=*stokvel\_id*))

*if* member\_role == 'admin' and admin\_count == 1:

flash("Cannot remove the last admin of the stokvel.")

*return* redirect(url\_for('view\_stokvel\_members', *stokvel\_id*=*stokvel\_id*))

*try*:

remove\_member\_query = """

DELETE FROM stokvel\_members WHERE id = %s AND stokvel\_id = %s

"""

support.execute\_query('insert', remove\_member\_query, (member\_to\_remove\_id, *stokvel\_id*))

flash("Member removed successfully!")

*except* Exception *as* e:

flash(f"Error removing member: {e}")

print(f"Error removing member: {e}")

*return* redirect(url\_for('view\_stokvel\_members', *stokvel\_id*=*stokvel\_id*))

@app.route('/stokvel/<int:stokvel\_id>/delete', *methods*=['POST'])

def **delete\_stokvel**(*stokvel\_id*):

*if* 'user\_id' not in session:

*return* redirect('/login')

user\_id = session['user\_id']

is\_admin\_query = """

SELECT 1 FROM stokvel\_members

WHERE stokvel\_id = %s AND user\_id = %s AND role = 'admin'

"""

is\_admin = support.execute\_query('search', is\_admin\_query, (*stokvel\_id*, user\_id))

*if* not is\_admin:

flash("You do not have permission to delete this stokvel.")

*return* redirect('/stokvels')

*try*:

delete\_transactions\_query = "DELETE FROM transactions WHERE stokvel\_id = %s"

support.execute\_query('insert', delete\_transactions\_query, (*stokvel\_id*,))

delete\_members\_query = "DELETE FROM stokvel\_members WHERE stokvel\_id = %s"

support.execute\_query('insert', delete\_members\_query, (*stokvel\_id*,))

delete\_stokvel\_query = "DELETE FROM stokvels WHERE id = %s"

support.execute\_query('insert', delete\_stokvel\_query, (*stokvel\_id*,))

flash("Stokvel deleted successfully!")

*except* Exception *as* e:

flash(f"Error deleting stokvel: {e}")

print(f"Error deleting stokvel: {e}")

*return* redirect('/stokvels')

@app.route('/payment\_methods')

def **payment\_methods**():

*if* 'user\_id' not in session:

*return* redirect('/login')

user\_id = session['user\_id']

query = """

SELECT id, user\_id, type, details, is\_default, created\_at

FROM payment\_methods

WHERE user\_id = %s

ORDER BY is\_default DESC, created\_at DESC

"""

payment\_methods\_data = support.execute\_query('search', query, (user\_id,))

payment\_methods\_list = []

columns = ['id', 'user\_id', 'type', 'details', 'is\_default', 'created\_at']

*if* payment\_methods\_data:

*for* row *in* payment\_methods\_data:

payment\_methods\_list.append(dict(zip(columns, row)))

*return* render\_template('payment\_methods.html', *payment\_methods*=payment\_methods\_list)

@app.route('/add\_payment\_method', *methods*=['POST'])

def **add\_payment\_method**():

*if* 'user\_id' not in session:

*return* redirect('/login')

user\_id = session['user\_id']

payment\_type = request.form.get('type')

details = request.form.get('details')

is\_default = request.form.get('is\_default') == 'true'

*try*:

*# First, update all existing payment methods to not be default*

*if* is\_default:

update\_query = """

UPDATE payment\_methods

SET is\_default = false

WHERE user\_id = %s

"""

support.execute\_query('insert', update\_query, (user\_id,))

*# Then insert the new payment method*

query = """

INSERT INTO payment\_methods (user\_id, type, details, is\_default)

VALUES (%s, %s, %s, %s)

RETURNING id

"""

result = support.execute\_query('insert', query, (user\_id, payment\_type, details, is\_default))

*if* result:

flash('Payment method added successfully!')

*else*:

flash('Error adding payment method: No ID returned')

*except* Exception *as* e:

flash(f'Error adding payment method: {e}')

print(f"Error adding payment method: {e}")

*return* redirect('/payment\_methods')

@app.route('/set\_default\_payment\_method', *methods*=['POST'])

def **set\_default\_payment\_method**():

*if* 'user\_id' not in session:

*return* redirect('/login')

user\_id = session['user\_id']

payment\_method\_id = request.form.get('payment\_method\_id')

*try*:

update\_query = """

UPDATE payment\_methods

SET is\_default = false

WHERE user\_id = %s

"""

support.execute\_query('insert', update\_query, (user\_id,))

set\_default\_query = """

UPDATE payment\_methods

SET is\_default = true

WHERE id = %s AND user\_id = %s

"""

support.execute\_query('insert', set\_default\_query, (payment\_method\_id, user\_id))

flash('Default payment method updated successfully!')

*except* Exception *as* e:

flash(f'Error updating default payment method: {e}')

print(f"Error updating default payment method: {e}")

*return* redirect('/payment\_methods')

@app.route('/delete\_payment\_method', *methods*=['POST'])

def **delete\_payment\_method**():

*if* 'user\_id' not in session:

*return* redirect('/login')

user\_id = session['user\_id']

payment\_method\_id = request.form.get('payment\_method\_id')

*try*:

check\_query = """

SELECT is\_default FROM payment\_methods

WHERE id = %s AND user\_id = %s

"""

result = support.execute\_query('search', check\_query, (payment\_method\_id, user\_id))

*if* result and result[0][0]:

flash('Cannot delete the default payment method. Please set another payment method as default first.')

*return* redirect('/payment\_methods')

delete\_query = """

DELETE FROM payment\_methods

WHERE id = %s AND user\_id = %s

"""

support.execute\_query('insert', delete\_query, (payment\_method\_id, user\_id))

flash('Payment method deleted successfully!')

*except* Exception *as* e:

flash(f'Error deleting payment method: {e}')

print(f"Error deleting payment method: {e}")

*return* redirect('/payment\_methods')

@app.route('/settings')

@login\_required

def **settings**():

*# Get user settings from database*

user\_settings = get\_user\_settings(session['user\_id'])

*return* render\_template('settings.html', *user*=user\_settings)

@app.route('/settings/update', *methods*=['POST'])

@login\_required

def **update\_settings**():

*try*:

data = request.get\_json()

section = data.get('section')

setting = data.get('setting')

value = data.get('value')

*# Update setting in database*

success = update\_user\_setting(session['user\_id'], section, setting, value)

*return* jsonify({'success': success})

*except* Exception *as* e:

print(f"Error updating settings: {str(e)}")

*return* jsonify({'success': False, 'error': str(e)})

def **get\_user\_settings**(*user\_id*):

*try*:

conn = support.db\_connection()

cur = conn.cursor()

*# Get user profile information*

cur.execute("""

SELECT full\_name, email, phone, profile\_picture,

two\_factor\_enabled, reminders\_enabled,

email\_notifications, sms\_notifications, weekly\_summary,

dark\_mode, remember\_me

FROM users

WHERE id = %s

""", (*user\_id*,))

user\_data = cur.fetchone()

*if* user\_data:

*return* {

'full\_name': user\_data[0],

'email': user\_data[1],

'phone': user\_data[2],

'profile\_picture': user\_data[3],

'two\_factor\_enabled': user\_data[4],

'reminders\_enabled': user\_data[5],

'email\_notifications': user\_data[6],

'sms\_notifications': user\_data[7],

'weekly\_summary': user\_data[8],

'dark\_mode': user\_data[9],

'remember\_me': user\_data[10]

}

*return* None

*except* Exception *as* e:

print(f"Error getting user settings: {str(e)}")

*return* None

*finally*:

*if* 'cur' in locals():

cur.close()

*if* 'conn' in locals():

conn.close()

def **update\_user\_setting**(*user\_id*, *section*, *setting*, *value*):

*try*:

conn = support.db\_connection()

cur = conn.cursor()

*# Map section and setting to database column*

setting\_map = {

'Profile Settings': {

'Full Name': 'full\_name',

'Email': 'email',

'Phone Number': 'phone',

'Profile Picture': 'profile\_picture'

},

'Account Security': {

'Two-Factor Authentication': 'two\_factor\_enabled'

},

'Group Preferences': {

'Contribution Reminders': 'reminders\_enabled'

},

'Notifications': {

'Email Notifications': 'email\_notifications',

'SMS Notifications': 'sms\_notifications',

'Weekly Summary': 'weekly\_summary'

},

'App Preferences': {

'Dark Mode': 'dark\_mode',

'Remember Me': 'remember\_me'

}

}

column = setting\_map.get(*section*, {}).get(*setting*)

*if* not column:

*return* False

*# Update the setting in the database*

cur.execute(f"""

UPDATE users

SET {column} = %s

WHERE id = %s

""", (*value*, *user\_id*))

conn.commit()

*return* True

*except* Exception *as* e:

print(f"Error updating user setting: {str(e)}")

*return* False

*finally*:

*if* 'cur' in locals():

cur.close()

*if* 'conn' in locals():

conn.close()

@app.route('/profile')

@login\_required

def **profile**():

user\_id = session['user\_id']

query = "SELECT username, email FROM users WHERE id = %s"

user\_data = support.execute\_query("search", query, (user\_id,))

*if* user\_data:

user = {

'username': user\_data[0][0],

'email': user\_data[0][1]

}

*return* render\_template('profile.html', *user*=user)

*else*:

flash("Error fetching user data")

*return* redirect('/home')

@app.route('/profile/update', *methods*=['POST'])

@login\_required

def **update\_profile**():

user\_id = session['user\_id'] *# This is now firebase\_uid*

username = request.form.get('username')

email = request.form.get('email')

*# Password change is now handled by Firebase client-side or specific Firebase Admin SDK methods*

*try*:

*# Update Firebase user profile*

auth.update\_user(

user\_id,

*email*=email,

*display\_name*=username

)

*# Update username and email in your local PostgreSQL database*

*# Removed password field from update*

query = "UPDATE users SET username = %s, email = %s WHERE firebase\_uid = %s"

support.execute\_query('insert', query, (username, email, user\_id))

flash("Profile updated successfully!")

*return* redirect('/profile')

*except* auth.AuthError *as* e:

print(f"Firebase Profile Update error: {e.code} - {e.message}")

flash(f"Error updating profile in Firebase: {e.message}")

*return* redirect('/profile')

*except* Exception *as* e:

flash(f"Error updating profile in local database: {str(e)}")

*return* redirect('/profile')

@app.context\_processor

def **inject\_user\_name**():

user\_name = None

*if* 'user\_id' in session:

*# Changed to firebase\_uid*

user\_query = "SELECT username FROM users WHERE firebase\_uid = %s"

user\_data = support.execute\_query("search", user\_query, (session['user\_id'],))

*if* user\_data:

user\_name = user\_data[0][0]

*return* dict(*user\_name*=user\_name)

@app.route('/pricing')

def **pricing**():

*if* 'user\_id' in session:

cursor = mysql.connection.cursor(*dictionary*=True)

cursor.execute("SELECT name FROM users WHERE id = %s", (session['user\_id'],))

user = cursor.fetchone()

cursor.close()

*return* render\_template('pricing.html', *user\_name*=user['name'] *if* user *else* None)

*return* render\_template('pricing.html', *user\_name*=None)

@app.route('/chat', *methods*=['POST'])

@login\_required

def **handle\_chat**():

*try*:

data = request.get\_json()

*if* not data or 'message' not in data:

*return* jsonify({'error': 'No message provided'}), 400

user\_message = data['message'].lower()

user\_id = session.get('user\_id')

*if* not user\_id:

*return* jsonify({'error': 'User not logged in'}), 401

*# Get user context from database*

*with* support.db\_connection() *as* conn:

*with* conn.cursor() *as* cur:

*# Get user info*

cur.execute("SELECT username FROM users WHERE id = %s", (user\_id,))

user = cur.fetchone()

*if* not user:

*return* jsonify({'error': 'User not found'}), 404

username = user[0]

*# Get total savings*

cur.execute("""

SELECT SUM(amount) FROM transactions

WHERE user\_id = %s AND type = 'contribution' AND status = 'completed'

""", (user\_id,))

total\_saved = cur.fetchone()[0] or 0

*# Get stokvel memberships*

cur.execute("""

SELECT s.name, s.monthly\_contribution, s.target\_date

FROM stokvels s

JOIN stokvel\_members sm ON s.id = sm.stokvel\_id

WHERE sm.user\_id = %s

""", (user\_id,))

stokvels = cur.fetchall()

*# Get recent transactions*

cur.execute("""

SELECT amount, type, description, transaction\_date

FROM transactions

WHERE user\_id = %s

ORDER BY transaction\_date DESC

LIMIT 5

""", (user\_id,))

transactions = cur.fetchall()

*# Simple rule-based response system*

response = "I'm not sure how to help with that. You can ask me about your savings, stokvels, or recent transactions."

*if* "how much" in user\_message and "saved" in user\_message:

response = f"You have saved R{total\_saved} in total across all your stokvels."

*elif* "stokvel" in user\_message:

*if* stokvels:

stokvel\_list = "\n".join([f"- {s[0]}: R{s[1]} monthly" *for* s *in* stokvels])

response = f"You are a member of these stokvels:\n{stokvel\_list}"

*else*:

response = "You are not currently a member of any stokvels."

*elif* "recent" in user\_message and "transaction" in user\_message:

*if* transactions:

transaction\_list = "\n".join([f"- {t[1]}: R{t[0]} ({t[2]}) on {t[3]}" *for* t *in* transactions])

response = f"Your recent transactions:\n{transaction\_list}"

*else*:

response = "You don't have any recent transactions."

*elif* "hello" in user\_message or "hi" in user\_message:

response = f"Hello {username}! How can I help you today? You can ask me about your savings, stokvels, or recent transactions."

*return* jsonify({'response': response})

*except* Exception *as* e:

print(f"Error in handle\_chat: {str(e)}")

*return* jsonify({'error': 'Internal server error'}), 500

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

app.run(*debug*=True, *host*='0.0.0.0', *port*=8080)

/etc/postgresql/10/main/pg\_hba.conf

(1 row)