CODE DOCUMENTATION

SEPTEMBER 28,2024

1 . AharSetu Web Application Documentation (Python Flask App)

1.0.1 Overview

AharSetu is a food donation platform that connects donors, such as restaurants or individuals, with beneficiaries, including orphanages, NGOs, or people in need. Built using **Flask**, the platform allows users to manage food posts, search for available food donations by city, and reserve them. Donors can create food posts with details about the food and its availability, while beneficiaries can reserve the posts. The project includes user authentication, allowing both donors and beneficiaries to register, log in, and manage their profiles. It also features a password reset function via email for beneficiaries.

AharSetu integrates **MySQL** as the database to store user, food post, and feedback information. **SQLAlchemy** is used as the ORM for database interactions, while **Celery** is employed for background task processing, such as sending email notifications to donors when their food is reserved and automatically deleting expired posts after 24 hours. Feedback submission is also supported, allowing users to rate and comment on their experience. With the help of **Flask-Mail**, email notifications are seamlessly sent in the background. The platform offers a simple, user-friendly interface built using **HTML, CSS, and JavaScript**, with dynamic content rendered via **Jinja templates**.

1.1 Libraries Overview

```
from flask import Flask, jsonify, render_template, request, redirect, url_for, g, session, flash
import pymysql
import secrets
import pymysql.cursors
from werkzeug.security import generate_password_hash, check_password_hash
from flask_mail import Mail, Message
from dotenv import load_dotenv
from datetime import datetime, timedelta

```

Flask:

- Flask: Creates the Flask web application instance.
- jsonify: Converts Python dictionaries into JSON format for API responses.
- render_template: Renders HTML templates for dynamic web page generation.
- request: Retrieves data from incoming HTTP requests (e.g., form inputs, query parameters).
- redirect: Redirects the user to a different URL after an action.
- url_for: Generates dynamic URLs for defined routes.
- session: Manages user session data (like login state).
- flash: Sends temporary messages to the user (e.g., success or error notifications).

PyMySQL:

- pymysql: Connects to the MySQL database and executes SQL queries.
- pymysql.cursors: Provides cursor classes for retrieving query results in different formats (e.g., dictionaries).

Secrets:

- secrets: Generates secure random tokens for session management and password resets.

Werkzeug Security:

- generate_password_hash: Hashes plain text passwords for secure storage in the database.
- check_password_hash: Verifies that a plain text password matches a stored hashed password.

Flask-Mail:

- Mail: Configures email sending functionality within the Flask app.
- Message: Creates email messages for notifications (e.g., confirmation emails).

Dotenv:

- load_dotenv: Loads environment variables from a .env file to manage sensitive configuration securely.

1.2 Application Setup & Database Connection Setup

```
16  app = Flask(__name__)
17  app.secret_key = "helloworld"  # Secret key for session management
18
19  # MySQL Database connection
20  db = pymysql.connect(
21      host="localhost",
22      user="root",
23      password="9666099560",
24      database="mydatabase"
25  )
```

app = Flask(__name__)

- **Application**: Initializes a new instance of the Flask web application.
- **Overview**: This line creates the main application object (app) that allows you to define routes, handle requests, and manage configurations. The __name__ parameter helps Flask understand where to find templates and static files relative to the location of the code.

---

app.secret_key = "helloworld"

- **Application**: Sets a secret key for the Flask application to manage sessions securely.
- **Overview**: The secret key is essential for signing session cookies, providing a layer of security against cookie tampering. It is crucial for maintaining user sessions and ensuring that session data cannot be

easily forged. In production, a more complex key should be used for better security.

---

db = pymysql.connect(host="localhost", user="root", password="9666099560", database="mydatabase")

- **Application**: Establishes a connection to a MySQL database using the PyMySQL library.
- **Overview**: This snippet connects to a MySQL server running on localhost (the local machine) using the specified credentials (username and password) and selects the database named mydatabase. This connection allows the application to perform database operations, such as executing queries to store or retrieve user data, making it essential for applications that require persistent data storage.

1.3 Email Configuration

```
27  # Configure Flask-Mail
28  app.config['MAIL_SERVER'] = 'smtp.gmail.com'  # Mail server
29  app.config['MAIL_PORT'] = 587  # Mail port
30  app.config['MAIL_USERNAME'] = 'saideeprangoni634@gmail.com'
31  app.config['MAIL_PASSWORD'] = 'ivuv epqn vsov baay'
32  app.config['MAIL_USE_TLS'] = True
33  app.config['MAIL_USE_SSL'] = False
34  app.config['DEBUG'] = True
35
36  mail = Mail(app)
```

app.config['MAIL_SERVER'] = 'smtp.gmail.com'

- **Application**: Configures the mail server to Gmail's SMTP.
- **Overview**: Specifies the server used for sending emails.

---

app.config['MAIL_PORT'] = 587

- **Application**: Sets the port for the mail server.
- **Overview**: Port 587 is used for secure SMTP with TLS.

---

app.config['MAIL_USERNAME'] = 'saideeprangoni634@gmail.com'

- **Application**: Defines the email address for sending emails.
- **Overview**: Specifies the username for mail server authentication.

---

app.config['MAIL_PASSWORD'] = 'ivuv epqn vsov baay'

- **Application**: Sets the password for the email account.
- **Overview**: Provides authentication credentials for the mail server.

---

app.config['MAIL_USE_TLS'] = True

- **Application**: Enables TLS for email transmission.
- **Overview**: Ensures secure email communication with encryption.

---

app.config['MAIL_USE_SSL'] = False

- **Application**: Disables SSL for email connections.
- **Overview**: Uses TLS instead of SSL for enhanced security.

---

app.config['DEBUG'] = True

- **Application**: Enables debug mode for the application.
- **Overview**: Provides detailed error messages and auto-reloads during development.

---

mail = Mail(app)

- **Application**: Initializes the Flask-Mail extension.
- **Overview**: Integrates email functionality into the Flask app.

1.4 Application-Related Database Queries

Note: All the queries are written and exceuted in MySQL Workbench.

```
CREATE DATABASE mydatabase;

USE mydatabase;

CREATE TABLE beneficiary (
  id INT AUTO_INCREMENT PRIMARY KEY,
  username VARCHAR(255) NOT NULL,
  contact VARCHAR(20),
  email VARCHAR(255) NOT NULL,
  password VARCHAR(255) NOT NULL,
  city VARCHAR(100)
);

CREATE TABLE feedback (
  id INT AUTO_INCREMENT PRIMARY KEY,
  name VARCHAR(255) NOT NULL,
  rating INT,
  category ENUM('suggestion', 'issue', 'compliment') NOT NULL,
  feedback TEXT,
  created_at DATETIME DEFAULT CURRENT_TIMESTAMP
);

CREATE TABLE password_resets (
  id INT AUTO_INCREMENT PRIMARY KEY,
  email VARCHAR(255) NOT NULL,
  token VARCHAR(255) NOT NULL,
  created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
  expires_at TIMESTAMP
);

CREATE TABLE password_resetsb (
  id INT AUTO_INCREMENT PRIMARY KEY,
  email VARCHAR(255) NOT NULL,
```

```
  token VARCHAR(255) NOT NULL,
  created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
  expires_at TIMESTAMP
);

CREATE TABLE users (
  id INT AUTO_INCREMENT PRIMARY KEY,
  username VARCHAR(255) NOT NULL,
  contact VARCHAR(15),
  email VARCHAR(255) NOT NULL,
  password VARCHAR(255) NOT NULL,
  city VARCHAR(100),
  created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

CREATE TABLE food_posts (
  id INT AUTO_INCREMENT PRIMARY KEY,
  username VARCHAR(255),
  food_details VARCHAR(255),
  people_served INT,
  city VARCHAR(100),
  additional_notes TEXT,
  created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

1.5 Beneficiary Related Routings

1.5.1 Registration,login,landing Pages

```python
@app.route('/beneficiary_registration', methods=['GET', 'POST'])
def beneficiary_registration():
    if request.method == 'POST':
        username = request.form.get('username')
        contact = request.form.get('contact')
        email = request.form.get('email')
        password = request.form.get('password')
        confirm_password = request.form.get('confirm_password')
        city = request.form.get('cityField')

        # Validate password
        if password != confirm_password:
            flash('Passwords do not match!', 'danger')
            return redirect(url_for('beneficiary_registration'))

        # Insert beneficiary into the database without hashing (consider hashing in production)
        try:
            cursor = db.cursor()
            insert_query = """
                INSERT INTO beneficiary (username, contact, email, password, city)
                VALUES (%s, %s, %s, %s, %s)
            """
            cursor.execute(insert_query, (username, contact, email, password, city))  # Storing raw password
            db.commit()
            cursor.close()

            flash('Registration successful!', 'success')
            return redirect(url_for('beneficiary_login'))

        except Exception as err:
            db.rollback()
            flash(f'Database error: {str(err)}', 'danger')
            return redirect(url_for('beneficiary_registration'))

    return render_template('beneficiary_registration.html')
```

- **URL:** /beneficiary_registration
- **Method:** GET, POST

Overview

This route handles the registration of new beneficiaries:

- **GET:** Renders the registration form.
- **POST:** Validates the input, inserts the beneficiary into the database, and manages success or error messages.

Returns

- **GET Request:** Renders the registration form (HTML).

- **POST Request:**
    - On success: Redirects to the login page with a success message.
    - On validation error (password mismatch): Redirects back to the form with an error message.
    - On database error: Redirects back to the form with an error message.

```python
@app.route('/beneficiary_login', methods=['GET', 'POST'])
def beneficiary_login():
    if request.method == 'POST':
        email = request.form.get('email')
        password = request.form.get('password')  # Plain text password comparison

        try:
            # Cursor setup to execute MySQL commands
            with db.cursor(pymysql.cursors.DictCursor) as cursor:
                # Query to fetch beneficiary by email
                sql = "SELECT * FROM beneficiary WHERE email = %s"
                cursor.execute(sql, (email,))
                beneficiary = cursor.fetchone()

                # Check if the beneficiary exists and compare the plain text password
                if beneficiary and beneficiary['password'] == password:
                    session['beneficiary_id'] = beneficiary['id']  # Storing session for beneficiary
                    session['email'] = beneficiary['email']
                    flash('Login successful!', 'success')
                    return redirect(url_for('landingpage_beneficiary'))  # Redirect to beneficiary landing page
                else:
                    flash('Invalid email or password.', 'danger')
                    return redirect(url_for('beneficiary_login'))

        except Exception as e:
            flash(f"An error occurred: {str(e)}", 'danger')
            return redirect(url_for('beneficiary_login'))

    return render_template('beneficiary_login.html')
```

Route Overview

- **URL:** /beneficiary_login
- **Method:** GET, POST

Overview

This route manages the login process for beneficiaries:

- **GET:** Renders the login form.
- **POST:** Validates credentials and manages session creation or error messages.

Returns

- **GET Request:** Renders the login form (HTML).

- **POST Request:**
    - On successful login: Redirects to the beneficiary landing page with a success message.
    - On invalid credentials: Redirects back to the form with an error message.
    - On exceptions: Redirects back to the form with an error message.

```python
@app.route('/landingpage_beneficiary', methods=['GET', 'POST'])
def landingpage_beneficiary():
    # Check if the beneficiary is logged in using beneficiary_id
    if 'beneficiary_id' not in session:
        return redirect(url_for('beneficiary_login'))

    return render_template('landingpage_beneficiary.html')


```

- **URL:** /landingpage_beneficiary
- **Method:** GET, POST

Overview

This route displays the landing page for beneficiaries after they log in. It checks if the beneficiary is authenticated by verifying the session.

Returns

- **GET Request:**
    - If the beneficiary is logged in, it renders the landing page (HTML).
    - If the beneficiary is not logged in, it redirects to the login page.

1.5.2 Search_for_food Routing

```python
@app.route('/search_for_food', methods=['GET', 'POST'])
def search_for_food():
    if 'beneficiary_id' not in session:
        return redirect(url_for('beneficiary_login'))
    if request.method == 'POST':
        city = request.form.get('city')
        beneficiary_id = session['beneficiary_id']
        try:
            conn = db
            with conn.cursor() as cursor:
                twenty_four_hours_ago = datetime.now() - timedelta(hours=24)

                query = """
                SELECT id, food_details, city, people_served, additional_notes
                FROM food_posts
                WHERE city = %s AND created_at > %s
                """
                cursor.execute(query, (city, twenty_four_hours_ago))
                food_posts = cursor.fetchall()
                results = [
                    {
                        "id": post[0],
                        "food_details": post[1],
                        "city": post[2],
                        "people_served": post[3],
                        "additional_notes": post[4]
                    }
                    for post in food_posts
                ]

                return jsonify({"food_posts": results})
        except pymysql.MySQLError as e:
            print(f"Database error: {e}")
            return jsonify({"error": "An error occurred while fetching data."}), 500

    return render_template('searchforfood.html')  # Render the search page directly for GET requests
```

- **URL:** /search_for_food
- **Methods:** GET, POST

Overview

This route allows beneficiaries to search for available food posts within the last 24 hours based on a specified city. It fetches food posts from the food_posts table that match the city and have been created within the past 24 hours. If the user is not logged in as a beneficiary, they are redirected to the login page.

Returns

- **POST Request:**
    - Returns a JSON object containing a list of food posts, with each post including:
        - id: The ID of the food post.
        - food_details: A description of the food available.

        - city: The city where the food is located.
        - people_served: The number of people the food can serve.
        - additional_notes: Any additional notes related to the post.
- **GET Request:**
    - Renders the searchforfood.html page, allowing the user to input the city and search for food posts.

Error Responses:

- **500 Internal Server Error:** If any database error occurs, it returns an error message detailing the issue.

1.5.3 View Food Posts Details Routing

```python
@app.route('/view_details/<int:post_id>', methods=['GET'])
def view_details(post_id):
    try:
        with db.cursor() as cursor:
            query = """SELECT u.username, u.contact, u.email
                FROM food_posts fp
                JOIN users u ON fp.username = u.email
                WHERE fp.id = %s"""
            cursor.execute(query, (post_id,))
            result = cursor.fetchone()
            if result:
                return jsonify({"username": result[0], "contact": result[1], "email": result[2]})
            else:
                return jsonify({"error": "Post not found."}), 404
    except Exception as e:
        return jsonify({"error": str(e)}), 500
```

- **URL:** /view_details/<int:post_id>
- **Method:** GET

Overview

This route retrieves and displays the details of a specific food post based on the provided post ID. It joins the food_posts and users tables to fetch the username, contact, and email of the user who posted the food.

Returns

- **Successful Response (200 OK):**
    - Returns a JSON object containing the username, contact, and email of the user associated with the specified food post.
- **Error Responses:**
    - **404 Not Found:** If no post is found with the given post ID, it returns an error message indicating the post was not found.
    - **500 Internal Server Error:** If any exception occurs during the database query, it returns an error message with the exception details.

1.5.3 Reserving The Food Post Routing

```python
@app.route('/reserve/<int:post_id>', methods=['GET'])
def reserve_food(post_id):
    try:
        with db.cursor() as cursor:
            query = """SELECT u.email
                       FROM food_posts fp
                       JOIN users u ON fp.username = u.email
                       WHERE fp.id = %s"""
            cursor.execute(query, (post_id,))
            result = cursor.fetchone()

            if result:
                donor_email = result[0]
                print(f"Donor email found: {donor_email}")  # Log found email

                # Send confirmation email to the donor
                send_email_to_donor(donor_email)

                return jsonify({"success": True})
            else:
                print("Post not found.")  # Log if post is not found
                return jsonify({"error": "Post not found."}), 404
    except Exception as e:
        print(f"Error reserving food: {e}")
        return jsonify({"error": str(e)}), 500

def send_email_to_donor(donor_email):
    subject = "Food Reservation Confirmation"
    body = "Someone has reserved the food you posted.They will contact you soon.thank you for your kindness...."

    msg = Message(subject, sender='saideeprangoni634@gmail.com', recipients=[donor_email])
    msg.body = body
    try:
        mail.send(msg)
        print("Email sent successfully")
    except Exception as e:
        print(f"Failed to send email: {e}")  # Log the error
```

- **URL:** /reserve/<int:post_id>
- **Method:** GET

Overview

This route allows a beneficiary to reserve a food post by its unique post_id. Upon reservation, the system retrieves the donor's email associated with the post and sends a confirmation email notifying the donor that someone has reserved their food. If the food post is not found, a 404 error is returned.

Returns

- **Success:**
    - A JSON object indicating a successful reservation ({"success": True}).
    - Sends a confirmation email to the donor associated with the reserved food post.
- **Error:**
    - If the food post is not found, returns a 404 error with the message: {"error": "Post not found."}
    - If there is an internal error, returns a 500 error with the error message:
      {"error": "<error_message>"}.

1.5.4 Beneficiary Profile Routing

```python
@app.route('/beneficiary_profile', methods=['GET', 'POST'])
def beneficiary_profile():
    beneficiary_id = session.get('beneficiary_id')
    if not beneficiary_id:
        flash('You need to log in first.', 'danger')
        return redirect(url_for('beneficiary_login'))

    # Fetch beneficiary data from the database
    cursor = db.cursor()
    cursor.execute("SELECT username, contact, email, city FROM beneficiary WHERE id = %s", (beneficiary_id,))
    beneficiary = cursor.fetchone()

    # Assigning the fetched data to a dictionary for rendering
    beneficiary = {
        'username': beneficiary[0],
        'contact': beneficiary[1],
        'email': beneficiary[2],
        'city': beneficiary[3]
    }

    if request.method == 'POST':
        current_password = request.form.get('current_password')
        new_password = request.form.get('new_password')

        # Check current password
        cursor.execute("SELECT password FROM beneficiary WHERE id = %s", (beneficiary_id,))
        stored_password = cursor.fetchone()[0]

        if stored_password == current_password:  # Adjust this line for hashed passwords
            cursor.execute("UPDATE beneficiary SET password = %s WHERE id = %s", (new_password, beneficiary_id))
            db.commit()
            flash('Password updated successfully!', 'success')
        else:
            flash('Current password is incorrect.', 'danger')

    return render_template('beneficiary_profile.html', beneficiary=beneficiary)
```

- **URL:** /beneficiary_profile
- **Method:** GET, POST

Overview

This route handles the beneficiary profile page, allowing beneficiaries to view their profile information and update their password. If the beneficiary is not logged in, they will be redirected to the login page. The page displays the beneficiary's username, contact, email, and city, and provides a form for changing the password.

Returns

- **GET Request:**
    - Renders the beneficiary_profile.html template, displaying the beneficiary's information.
- **POST Request:**
    - Checks the provided current password against the stored password in the database.

- If the password matches, updates the beneficiary's password in the database and displays a success message.
- If the password does not match, displays an error message.

1.5.5 Updating Password Routing

```python
@app.route('/update_beneficiary_password', methods=['POST'])
def update_beneficiary_password():
    beneficiary_id = session.get('beneficiary_id')
    if not beneficiary_id:
        flash('You need to log in first.', 'danger')
        return redirect(url_for('beneficiary_login'))

    current_password = request.form.get('current_password')
    new_password = request.form.get('new_password')

    # Logic to check the current password
    cursor = db.cursor()
    cursor.execute("SELECT password FROM beneficiary WHERE id = %s", (beneficiary_id,))
    result = cursor.fetchone()

    if result:
        stored_password = result[0]
        if stored_password == current_password:  # Adjust this line for hashed passwords
            cursor.execute("UPDATE beneficiary SET password = %s WHERE id = %s", (new_password, beneficiary_id))
            db.commit()
            flash('Password updated successfully!', 'success')
        else:
            flash('Current password is incorrect.', 'danger')
    else:
        flash('Beneficiary not found.', 'danger')

    return redirect(url_for('beneficiary_profile'))
```

- **URL:** /update_beneficiary_password
- **Method:** POST

Overview

This route allows a logged-in beneficiary to update their password. The beneficiary must provide their current password and the new password they wish to set. If the current password matches the stored password, the new password will be updated in the database.

Returns

- Redirects to the beneficiary profile page after attempting to update the password.

1.6 User/Donor Related Routings

1.6.1 User Registration,Login,Landing pages

```python
##################################################
### User related routings ####
@app.route('/user_registration', methods=['GET', 'POST'])
def user_registration():
    if request.method == 'POST':
        username = request.form['username']
        contact = request.form['contact']
        email = request.form['email']
        password = request.form['password']
        confirm_password = request.form['confirmPassword']
        city = request.form['cityField']

        # Simple password match validation
        if password != confirm_password:
            flash('Passwords do not match!', 'danger')
            return redirect(url_for('user_registration'))

        # Inserting raw password into the database without hashing
        try:
            with db.cursor() as cursor:
                sql = """
                INSERT INTO users (username, contact, email, password, city)
                VALUES (%s, %s, %s, %s, %s)
                """
                cursor.execute(sql, (username, contact, email, password, city))  # Storing raw password
                db.commit()
            flash('Account created successfully!', 'success')
            return redirect(url_for('user_login'))
        except Exception as e:
            print(e)
            flash('There was an issue creating your account. Please try again.', 'danger')
            return redirect(url_for('user_registration'))

    return render_template('user_registration.html')
```

URL: /user_registration

Methods:

- GET
- POST

Purpose:

To handle user registration by accepting user details and storing them in the database.

Functionality:

1. User Registration Form Submission:
    - Description:
        - The route checks if the request method is POST, indicating the form has been submitted.
    - Data Retrieval:

        - It retrieves the user input values: username, contact, email, password, confirm_password, and city.

2. Password Match Validation:
    - Description:
        - A simple validation checks if the password and confirm_password fields match.
    - Behavior:
        - If they do not match, a flash message is displayed, and the user is redirected back to the registration page.
3. Database Insertion:
    - Description:
        - If the passwords match, the route attempts to insert the user details into the database.
    - SQL Query Execution:
        - An SQL query is prepared to insert the data into the users table.
        - The raw password is stored in the database without hashing.
    - Behavior:
        - If the insertion is successful, a success message is flashed, and the user is redirected to the login page.
        - If an error occurs, it is printed to the console, a failure message is flashed, and the user is redirected back to the registration page.
4. Rendering the Registration Form:
    - Description:
        - If the request method is GET, the route renders the user_registration.html template to display the registration form.

Returns:

- GET Request:
    - Renders the user_registration.html template for user input.
- POST Request:
    - On successful registration: Redirects to the user_login page with a success message.
    - On validation failure or error: Redirects back to the registration page with an error message.

```python
@app.route('/user_login', methods=['GET', 'POST'])
def user_login():
    if request.method == 'POST':
        email = request.form.get('email')
        password = request.form.get('password')  # Plain text password comparison

        try:
            # Cursor setup to execute MySQL commands
            with db.cursor(pymysql.cursors.DictCursor) as cursor:
                # Query to fetch user by email
                sql = "SELECT * FROM users WHERE email = %s"
                cursor.execute(sql, (email,))
                user = cursor.fetchone()

                # Check if the user exists and compare the plain text password
                if user and user['password'] == password:
                    session['user_id'] = user['id']
                    session['email'] = user['email']
                    flash('Login successful!', 'success')
                    return redirect(url_for('landingpage_user'))  # Replace with your dashboard route
                else:
                    flash('Invalid email or password.', 'danger')
                    return redirect(url_for('user_login'))  # Back to login if credentials fail

        except Exception as e:
            flash(f"An error occurred: {str(e)}", 'danger')
            return redirect(url_for('user_login'))

    # If it's a GET request, render the login form
    return render_template('user_login.html')
```

URL: /user_login

Methods:

- GET
- POST

Purpose:

To handle user login by verifying user credentials and initiating a session.

Functionality:

1. Login Form Submission:

- Description:
  - The route checks if the request method is POST, indicating the login form has been submitted.
- Data Retrieval:
  - It retrieves the user input values: email and password.

2. User Authentication:
   - Description:
     - The route attempts to verify the user's credentials against the database.
   - Database Query Execution:
     - A cursor is set up to execute MySQL commands, and an SQL query fetches the user record based on the provided email.
   - Behavior:
     - If a user record is found and the stored password matches the entered password:
       - The user's ID and email are stored in the session.
       - A success message is flashed, and the user is redirected to the landingpage_user route (or your dashboard route).
     - If the credentials are invalid:
       - An error message is flashed, and the user is redirected back to the login page.
3. Error Handling:
   - Description:
     - Any exceptions raised during the process are caught.
   - Behavior:
     - An error message is flashed indicating an issue occurred, and the user is redirected back to the login page.
4. Rendering the Login Form:

- Description:
    - If the request method is GET, the route renders the user_login.html template to display the login form.

Returns:

- GET Request:
    - Renders the user_login.html template for user input.
- POST Request:
    - On successful login: Redirects to the landingpage_user page with a success message.
    - On invalid credentials or error: Redirects back to the login page with an error message.

```
358  @app.route('/landingpage_user')
359  def landingpage_user():
360      if 'user_id' not in session:
361          flash('You need to log in first.', 'warning')
362          return redirect(url_for('user_login'))
363      return render_template('landingpage_user.html')
```

URL: /landingpage_user

Methods:

- GET

Purpose:

To display the user's landing page after successful login while ensuring the user is authenticated.

Functionality:

1. User Authentication Check:
    - Description:
        - The route first checks if the user is logged in by verifying the presence of user_id in the session.
    - Behavior:

        - If user_id is not found in the session, a warning message is flashed indicating that the user needs to log in.
        - The user is then redirected to the user_login route to prompt for login credentials.

2. Rendering the Landing Page:
    - Description:
        - If the user is authenticated, the route proceeds to render the landingpage_user.html template.
    - Behavior:
        - The landing page is displayed to the user, allowing access to user-specific features or content.

Returns:

- GET Request:
    - If the user is not logged in: Redirects to the user_login page with a warning message.
    - If the user is logged in: Renders the landingpage_user.html template for the authenticated user.

1.6.2 User Profile

```python
@app.route('/profile', methods=['GET', 'POST'])
def profile():
    user_id = session.get('user_id')
    if not user_id:
        flash('You need to log in first.', 'danger')
        return redirect(url_for('user_login'))

    cursor = db.cursor()
    cursor.execute("SELECT username, contact, email, city FROM users WHERE id = %s", (user_id,))
    user = cursor.fetchone()
    user = {
        'username': user[0],
        'contact': user[1],
        'email': user[2],
        'city': user[3]
    }

    if request.method == 'POST':
        current_password = request.form.get('current_password')
        new_password = request.form.get('new_password')


        # Example:
        cursor.execute("SELECT password FROM users WHERE id = %s", (user_id,))
        stored_password = cursor.fetchone()[0]
        if stored_password == current_password:
            cursor.execute("UPDATE users SET password = %s WHERE id = %s", (new_password, user_id))
            db.commit()
            flash('Password updated successfully!', 'success')
        else:
            flash('Current password is incorrect.', 'danger')

    return render_template('user_profile.html', user=user)
```

URL: /profile

Methods:

- GET
- POST

Purpose:

To display the user's profile information and allow the user to update their password.

Functionality:

1. User Authentication Check:
    - Description:
        - The route checks if the user is logged in by retrieving the user_id from the session.
    - Behavior:

        - If user_id is not found, it flashes a danger message ("You need to log in first.") and redirects to the user_login page.

2. Fetching User Data:
    - Description:
        - If the user is authenticated, the route retrieves the user's details (username, contact, email, city) from the database based on the user_id.
    - Behavior:
        - The retrieved data is stored in a dictionary format for rendering in the profile template.
3. Password Update Logic:
    - Description:
        - If the request method is POST, the route retrieves the current and new passwords from the submitted form.
    - Behavior:
        - It checks the database for the stored password corresponding to the user.
        - If the stored password matches the provided current password, the new password is updated in the database, and a success message ("Password updated successfully!") is flashed.
        - If the current password does not match, a danger message ("Current password is incorrect.") is flashed.

Returns:

- GET Request:
    - If the user is not logged in: Redirects to the user_login page with a danger message.
    - If the user is logged in: Renders the user_profile.html template with the user's details.
- POST Request:

- If the current password is correct: Updates the password in the database and flashes a success message, then remains on the same profile page.
- If the current password is incorrect: Flashes a danger message indicating the issue, then remains on the same profile page.

1.6.3 User Update Passsword

```python
@app.route('/update_password', methods=['POST'])
def update_password():
    user_id = session.get('user_id')
    if not user_id:
        flash('You need to log in first.', 'danger')
        return redirect(url_for('user_login'))

    current_password = request.form.get('current_password')
    new_password = request.form.get('new_password')

    # Here, implement your logic to check the current password
    cursor = db.cursor()
    cursor.execute("SELECT password FROM users WHERE id = %s", (user_id,))
    result = cursor.fetchone()

    if result:
        stored_password = result[0]
        if stored_password == current_password:  # Adjust this line for hashed passwords
            cursor.execute("UPDATE users SET password = %s WHERE id = %s", (new_password, user_id))
            db.commit()
            flash('Password updated successfully!', 'success')
        else:
            flash('Current password is incorrect.', 'danger')
    else:
        flash('User not found.', 'danger')

    return redirect(url_for('profile'))
```

URL: /update_password

Methods:

- POST

Purpose:

To allow users to update their password after verifying their current password.

Functionality:

1. User Authentication Check:
    - Description:

        - The route checks if the user is logged in by retrieving the user_id from the session.
    - Behavior:
        - If user_id is not found, it flashes a danger message ("You need to log in first.") and redirects to the user_login page.
2. Password Retrieval:
    - Description:
        - The route retrieves the current and new passwords from the submitted form.
    - Behavior:
        - A database cursor is created, and a query is executed to fetch the stored password for the user based on user_id.
3. Password Update Logic:
    - Description:
        - The route checks if a result was found for the user.
    - Behavior:
        - If the user exists, it compares the stored password with the provided current password.
        - If they match, the new password is updated in the database, and a success message ("Password updated successfully!") is flashed.
        - If the passwords do not match, a danger message ("Current password is incorrect.") is flashed.
        - If the user is not found, a danger message ("User not found.") is flashed.

Returns:

- POST Request:
    - If the user is not logged in: Redirects to the user_login page with a danger message.

- If the user exists and the current password is correct: Updates the password in the database, flashes a success message, and redirects back to the profile page.
- If the current password is incorrect: Flashes a danger message indicating the issue and redirects back to the profile page.
- If the user is not found: Flashes a danger message indicating the user was not found and redirects back to the profile page

1.6.4 Donations Page Routing

```python
@app.route('/donations', methods=['GET'])
def donations():
    # Check if the user is logged in
    if 'user_id' not in session:
        flash('You need to log in first.', 'danger')
        return redirect(url_for('user_login'))
    user_id = session['user_id']

    try:
        conn = db
        with conn.cursor() as cursor:

            # Get timestamp for 24 hours ago
            twenty_four_hours_ago = datetime.now() - timedelta(hours=24)

            # Query to fetch food posts by the user in the last 24 hours
            query = """
            SELECT id, food_details, city, people_served, additional_notes, created_at
            FROM food_posts
            WHERE username = (SELECT email FROM users WHERE id = %s)
            AND created_at > %s
            ORDER BY created_at DESC
            """
            cursor.execute(query, (user_id, twenty_four_hours_ago))

            # Fetching the results
            columns = [column[0] for column in cursor.description]
            food_posts = [dict(zip(columns, row)) for row in cursor.fetchall()]

            # If no posts are found, show a message
            if not food_posts:
                flash('No food posts available in the last 24 hours.', 'info')
    except pymysql.MySQLError as e:
        print(f"Database error: {e}")
        return jsonify({'status': 'error', 'message': 'Database error occurred'}), 500
    # Render the template with the food posts or an empty list
    return render_template('donations.html', food_posts=food_posts)
```

URL: /donations

Methods:

- GET

Purpose:

To display food posts made by the user within the last 24 hours.

Functionality:

1. User Authentication Check:
    - Description:

        - The route checks if the user is logged in by looking for user_id in the session.
    - Behavior:
        - If user_id is not found, it flashes a danger message ("You need to log in first.") and redirects the user to the user_login page.
2. Database Query for Food Posts:
    - Description:
        - A database connection is established, and a cursor is created for executing queries.
        - The timestamp for 24 hours ago is calculated to filter food posts.
    - Behavior:
        - A SQL query fetches food posts made by the user (identified by their email) within the last 24 hours, ordering the results by creation time.
3. Fetching and Formatting Results:
    - Description:
        - The results are fetched from the database cursor.
    - Behavior:
        - If food posts are found, they are formatted into a list of dictionaries, where each dictionary represents a food post with relevant details.
4. Handling No Results:
    - Description:
        - If no food posts are found for the user.
    - Behavior:
        - A flash message is displayed ("No food posts available in the last 24 hours.").

5. Error Handling:
    - Description:
        - A try-except block is used to catch any database-related errors.
    - Behavior:
        - If a MySQLError occurs, it prints the error to the console and returns a JSON response indicating a database error.

Returns:

- GET Request:
    - If the user is not logged in: Redirects to the user_login page with a danger message.
    - If food posts are found: Renders the donations.html template with the list of food posts.
    - If no food posts are found: Renders the donations.html template with an empty list and a flash message.
    - If a database error occurs: Returns a JSON response with an error message and status code 500.

1.6.5 Posting Food Post , Managing Food Post & Thank you Routings

```python

@app.route('/post_availability', methods=['POST'])
def post_availability():
    if request.method == 'POST':
        food_details = request.form.get('food_details')
        people_served = request.form.get('people_served')
        city = request.form.get('city')
        additional_notes = request.form.get('additional_notes')

        # Get username from the session (ensure the user is logged in)
        username = session.get('email')

        if not username:  # Check if the user is logged in
            flash('You need to log in first.', 'danger')
            return redirect(url_for('user_login'))

        try:
            cursor = db.cursor()
            query = """
                INSERT INTO food_posts (username, food_details, people_served, city, additional_notes)
                VALUES (%s, %s, %s, %s, %s)
            """
            cursor.execute(query, (username, food_details, people_served, city, additional_notes))
            db.commit()
            flash('Food availability posted successfully!', 'success')

            # Redirect to the thank you page after successful insert
            return redirect(url_for('thank_you'))
        except Exception as e:
            db.rollback()
            print(f"Database Error: {str(e)}")  # Log the detailed error
            flash('Error posting food availability: {}'.format(str(e)), 'danger')

        return redirect(url_for('landingpage_user'))
```

URL: /post_availability

Methods:

- POST

Purpose:

To allow logged-in users to post the availability of food.

Functionality:

1. Form Data Retrieval:
    - Description:
        - The route retrieves form data submitted via a POST request, including food_details, people_served, city, and additional_notes.
    - Behavior:

        - Data is collected from the request using request.form.get().
2. User Authentication Check:
    - Description:
        - The route checks if the user is logged in by fetching the email from the session.
    - Behavior:
        - If username (email) is not found, it flashes a danger message ("You need to log in first.") and redirects the user to the user_login page.
3. Database Insertion:
    - Description:
        - A database cursor is created, and an SQL query is prepared to insert a new food post into the food_posts table.
    - Behavior:
        - The cursor executes the insert query with the collected data, and changes are committed to the database.
4. Success Handling:
    - Description:
        - Upon successful insertion of the food post.
    - Behavior:
        - A flash message ("Food availability posted successfully!") is displayed, and the user is redirected to the thank_you page.
5. Error Handling:
    - Description:
        - A try-except block is used to handle any errors that may occur during database operations.
    - Behavior:

- If an error occurs, the transaction is rolled back, and a detailed error message is printed to the console. A flash message indicating the error is also displayed.
- The user is redirected back to the landingpage_user.

Returns:

- POST Request:
  - If the user is not logged in: Redirects to the user_login page with a danger message.
  - If the food post is successfully inserted: Redirects to the thank_you page with a success message.
  - If an error occurs during insertion: Redirects back to the landingpage_user with an error message.

```python
@app.route('/delete_post/<int:post_id>', methods=['POST'])
def delete_post(post_id):
    if 'user_id' not in session:
        flash('You need to log in first.', 'danger')
        return redirect(url_for('user_login'))

    user_id = session['user_id']

    try:
        conn = db  # Ensure db is a valid connection object
        with conn.cursor() as cursor:  # Using 'with' to ensure the cursor is closed automatically

            # Delete the post only if it belongs to the logged-in user
            cursor.execute("""
                DELETE FROM food_posts
                WHERE id = %s AND username = (SELECT email FROM users WHERE id = %s)
            """, (post_id, user_id))

            deleted_rows = cursor.rowcount
            conn.commit()

            if deleted_rows > 0:
                flash('Post deleted successfully!', 'success')
            else:
                flash('Post not found or you do not have permission to delete it.', 'danger')

    except pymysql.MySQLError as e:
        print(f"Database error: {e}")
        flash('An error occurred while deleting the post. Please try again.', 'danger')
        return jsonify({'status': 'error', 'message': 'Database error occurred'}), 500

    # Redirect to the dashboard page after deletion
    return redirect(url_for('landingpage_user'))
```

URL: /delete_post/<int:post_id>

Methods:

- POST

Purpose:

To allow users to delete a specific food post that they have created.

Functionality:

1. User Authentication Check:
    - Description:
        - The route checks if the user is logged in by verifying the presence of user_id in the session.
    - Behavior:
        - If the user is not logged in, a flash message ("You need to log in first.") is displayed, and the user is redirected to the user_login page.
2. Database Deletion:
    - Description:
        - A database connection and cursor are created to execute the deletion operation.
    - Behavior:
        - The SQL DELETE query attempts to remove the food post from the food_posts table where the id matches the provided post_id and the username matches the logged-in user's email (fetched from the users table).
    - Success Handling:
        - The number of deleted rows is checked using cursor.rowcount.
        - If rows were deleted, a success message ("Post deleted successfully!") is flashed.
        - If no rows were deleted, indicating that the post does not exist or the user does not have permission to delete it, an appropriate danger message is flashed.
3. Error Handling:
    - Description:

        - A try-except block is used to catch any database-related errors during the deletion process.
    - Behavior:
        - If a pymysql.MySQLError occurs, it prints the error and flashes a user-friendly error message.
        - In case of an error, the function returns a JSON response indicating an error, with a status code of 500.
4. Redirection:
    - Description:
        - After attempting to delete the post, the user is redirected to the landingpage_user, regardless of whether the deletion was successful or not.

```
502  @app.route('/thank_you')
503  def thank_you():
504      return render_template('thank_you.html')
505
```

URL: /thank_you

Methods:

- GET

Purpose:

To render a thank-you page after a successful action, such as posting food availability.

Functionality:

1. Rendering the Template:
    - Description:
        - When a user accesses the /thank_you URL, the route renders the thank_you.html template.
    - Behavior:

- This template can display a thank-you message or any additional information you'd like to provide to the user after they successfully posted food availability.

1.7.0 Forgot Password,Reset Pssword,Update Password Routings

1.7.1 For Users/Donors Routings

```python
@app.route('/forgot-password', methods=['GET', 'POST'])
def forgot_password():
    cursor = None  # Initialize cursor to None to avoid UnboundLocalError

    if request.method == 'POST':
        email = request.form.get('email')

        try:
            # Establish a database connection
            conn = db  # Ensure this points to your database connection
            cursor = conn.cursor()

            # Check if the email exists in the users table
            cursor.execute("SELECT * FROM users WHERE email = %s", (email,))
            user = cursor.fetchone()

            if user:
                # Generate a random token for password reset
                token = secrets.token_urlsafe(20)

                # Store the reset token in the password_resets table
                cursor.execute(
                    "INSERT INTO password_resets (email, token, expires_at) VALUES (%s, %s, NOW() + INTERVAL 1 HOUR)",
                    (email, token)
                )
                conn.commit()

                # Construct the reset URL
                reset_url = url_for('reset_password', token=token, _external=True)

                # Create the email message
                msg = Message(
                    subject="Password Reset Request",
                    recipients=[email],
                    sender=app.config['MAIL_USERNAME']  # Ensure this is configured correctly
                )
                msg.body = f"To reset your password, visit the following link: {reset_url}"

                # Try to send the email
                try:
                    mail.send(msg)
                    flash('A password reset link has been sent to your email.', 'success')
                    return redirect(url_for('user_login'))
                except Exception as e:
                    app.logger.error(f"Error sending email: {e}")
                    conn.rollback()  # Rollback in case of email sending failure
                    flash('An error occurred while sending the email. Please try again.', 'danger')
            else:
                flash('Email not found.', 'danger')

        except pymysql.MySQLError as err:
            app.logger.error(f"Database Error: {err}")
            flash('An error occurred during the request. Please try again later.', 'danger')
        finally:
            if cursor:
                cursor.close()
            if conn:
                conn.close()

    return render_template('forgot_password.html')

```

URL: /forgot-password

Methods:

- GET: To display the forgot password form.
- POST: To handle the submission of the email and initiate the password reset process.

Functionality:

1. Handling the GET Request:
    - Displays the forgot_password.html template where users can enter their email address to receive a password reset link.
2. Handling the POST Request:
    - Retrieves the email from the form submission.
    - Connects to the database and checks if the email exists in the users table.
    - If the email exists:
        - Generates a secure token using secrets.token_urlsafe(20).
        - Inserts the token into a password_resets table (ensure this table exists and is set up to store email, token, and expiration time).
        - Constructs a reset URL that includes the token.
        - Creates an email message with the reset link.
        - Attempts to send the email and displays appropriate success or error messages.
    - If the email does not exist, flashes a message indicating that the email is not found.
    - Handles database errors and ensures resources are closed properly.

```python
@app.route('/reset-password/<token>', methods=['GET', 'POST'])
def reset_password(token):
    conn = None
    cursor = None

    if request.method == 'POST':
        new_password = request.form.get('password')

        # Validate the new password
        if not new_password:
            flash('Password cannot be empty.', 'danger')
            return render_template('reset_password.html', token=token)

        try:
            # Establish a database connection
            conn = db  # Ensure this points to your database connection
            cursor = conn.cursor()

            # Verify the reset token exists and is valid (within expiration time)
            cursor.execute("SELECT email FROM password_resets WHERE token = %s AND expires_at > NOW()", (token,))
            result = cursor.fetchone()

            if result:
                email = result[0]  # Get the email from the tuple

                # Update the user's password based on the email (without hashing)
                cursor.execute(
                    "UPDATE users SET password = %s WHERE email = %s",
                    (new_password, email)  # Store the new password directly
                )
                conn.commit()

                # Delete the token after the password reset is successful
                cursor.execute("DELETE FROM password_resets WHERE token = %s", (token,))
                conn.commit()

                flash('Your password has been reset. Please log in.', 'success')
                return redirect(url_for('user_login'))
            else:
                flash('Invalid or expired token.', 'danger')

        except pymysql.MySQLError as err:
            app.logger.error(f"Database Error: {err}")
            flash('An error occurred during the password reset process.', 'danger')

        finally:
            # Close the cursor and connection safely
            if cursor:
                cursor.close()
            if conn:
                conn.close()

    return render_template('reset_password.html', token=token)
```

URL: /reset-password/<token>

Methods:

- GET: To display the password reset form.
- POST: To handle the submission of the new password and update it in the database.

Functionality:

1. Handling the GET Request:
    - Displays the reset_password.html template with a token to validate the password reset.
2. Handling the POST Request:

- Retrieves the new password from the form.
- Validates that the new password is not empty.
- Connects to the database and verifies if the reset token is valid and not expired.
- If valid:
  - Updates the user's password in the users table. *Note: It's important to hash the password before storing it.*
  - Deletes the reset token from the password_resets table to prevent reuse.
  - Displays a success message and redirects the user to the login page.
- If the token is invalid or expired, flashes an appropriate error message.
- Handles database errors and ensures resources are closed properly.

```python
@app.route('/update_password_', methods=['GET', 'POST'])
def update_password_():
    if request.method == 'POST':
        email = request.form.get('email')
        new_password = request.form.get('new_password')

        # Connect to the database
        connection = db
        try:
            with connection.cursor() as cursor:
                # Update the user's password in the 'users' table
                update_query = "UPDATE users SET password=%s WHERE email=%s"
                cursor.execute(update_query, (new_password, email))
                connection.commit()

                if cursor.rowcount == 0:
                    flash('No user found with this email address.', 'warning')
                else:
                    flash('Password updated successfully!', 'success')

        except Exception as e:
            flash('An error occurred while updating the password. Please try again.', 'danger')
        finally:
            connection.close()

        return redirect(url_for('user_login'))  # Redirect to login or another page as needed

    return render_template('update_password.html')  # Render the update password form
```

URL: /update_password_

Methods:

- GET: To display the password update form.
- POST: To handle the submission of the new password and update it in the database.

Functionality:

1. Handling the GET Request:
    - Displays the update_password.html template with a form for the user to input their new password.
2. Handling the POST Request:
    - Retrieves the user's email and the new password from the form.
    - Connects to the database and executes an update query to change the user's password in the users table.
    - If the email does not exist, it flashes a warning message.
    - If the update is successful, it flashes a success message.
    - Any exceptions during the database operation will result in an error message being flashed.
    - Finally, it redirects the user to the login page.

1.7.2 For Beneficiary Routings

```python
@app.route('/forgot-password-beneficiary', methods=['GET', 'POST'])
def forgot_password_beneficiary():
    if request.method == 'POST':
        email = request.form.get('email')

        conn = None
        cursor = None

        try:
            # Establish a database connection
            conn = db  # Ensure this points to your database connection
            cursor = conn.cursor()

            # Check if the email exists in the beneficiary table
            cursor.execute("SELECT * FROM beneficiary WHERE email = %s", (email,))
            beneficiary = cursor.fetchone()

            if beneficiary:
                # Generate a random token for password reset
                token = secrets.token_urlsafe(20)

                # Store the reset token in the password_resets table
                cursor.execute(
                    "INSERT INTO password_resetsb (email, token, expires_at) VALUES (%s, %s, NOW() + INTERVAL 1 HOUR)",
                    (email, token)
                )
                conn.commit()

                # Construct the reset URL
                reset_url = url_for('reset_password_beneficiary', token=token, _external=True)

                # Create the email message
                msg = Message(
                    subject="Password Reset Request for Beneficiary",
                    recipients=[email],
                    sender=app.config['MAIL_USERNAME']  # Ensure this is configured correctly
                )
                msg.body = f"To reset your password, visit the following link: {reset_url}"

                # Try to send the email
                try:
                    mail.send(msg)
                    flash('A password reset link has been sent to your email.', 'success')
                    return redirect(url_for('beneficiary_login'))
                except Exception as e:
                    app.logger.error(f"Error sending email: {e}")
                    conn.rollback()  # Rollback in case of email sending failure
                    flash('An error occurred while sending the email. Please try again.', 'danger')
            else:
                flash('Email not found.', 'danger')

        except pymysql.MySQLError as err:
            app.logger.error(f"Database Error: {err}")
            flash('An error occurred during the request. Please try again later.', 'danger')
        finally:
            # Ensure the cursor and connection are closed properly
            if cursor is not None:
                cursor.close()
            if conn is not None:
                conn.close()

    return render_template('forgot_password_beneficiary.html')
```

URL: /forgot-password-beneficiary

Methods:

- GET: To display the password reset form.
- POST: To handle the submission of the email for the password reset.

Functionality:

1. Handling the GET Request:
    - Displays the forgot_password_beneficiary.html template with a form for the user to input their email.
2. Handling the POST Request:
    - Retrieves the email from the form.
    - Establishes a database connection to check if the email exists in the beneficiary table.
    - If the email is found, it generates a random token for the password reset and stores it in the password_resetsb table.
    - Constructs a reset URL and sends a password reset email.
    - If the email is not found, it flashes a message indicating this.
    - Handles any database errors appropriately.

```python
@app.route('/reset-password-beneficiary/<token>', methods=['GET', 'POST'])
def reset_password_beneficiary(token):
    conn = None
    cursor = None

    if request.method == 'POST':
        new_password = request.form.get('password')

        # Validate the new password
        if not new_password:
            flash('Password cannot be empty.', 'danger')
            return render_template('reset_password_beneficiary.html', token=token)

        try:
            # Establish a database connection
            conn = db  # Ensure this points to your database connection
            cursor = conn.cursor()

            # Verify the reset token exists and is valid (within expiration time)
            cursor.execute("SELECT email FROM password_resetsb WHERE token = %s AND expires_at > NOW()", (token,))
            result = cursor.fetchone()

            if result:
                email = result[0]  # Get the email from the tuple

                # Update the beneficiary's password based on the email (without hashing)
                cursor.execute(
                    "UPDATE beneficiary SET password = %s WHERE email = %s",
                    (new_password, email)  # Store the new password directly
                )
                conn.commit()

                # Check if the password update was successful
                if cursor.rowcount == 0:
                    flash('No beneficiary found with that email.', 'danger')
                    return render_template('reset_password_beneficiary.html', token=token)
```

```
798                 
799                     # Delete the token after the password reset is successful
800                     cursor.execute("DELETE FROM password_resetsb WHERE token = %s", (token,))
801                     conn.commit()
802 
803                     flash('Your password has been reset. Please log in.', 'success')
804                     return redirect(url_for('beneficiary_login'))
805                 else:
806                     flash('Invalid or expired token.', 'danger')
807 
808             except pymysql.MySQLError as err:
809                 app.logger.error(f"Database Error: {err}")
810                 flash('An error occurred during the password reset process.', 'danger')
811 
812             finally:
813                 # Close the cursor and connection safely
814                 if cursor:
815                     cursor.close()
816                 if conn and conn.open:  # Check if the connection is still open before closing
817                     conn.close()
818 
819     return render_template('reset_password_beneficiary.html', token=token)
820 
```

URL: /reset-password-beneficiary/<token>

Methods:

- GET: To display the password reset form.
- POST: To process the new password and update it in the database.

Functionality:

1. Handling the GET Request:
    - Renders the reset_password_beneficiary.html template with the token to allow users to enter a new password.
2. Handling the POST Request:
    - Retrieves the new password from the form.
    - Validates that the new password is not empty.
    - Establishes a database connection and verifies if the provided token exists and is still valid.
    - If valid, updates the password in the beneficiary table based on the email retrieved from the token.
    - Deletes the token after a successful password reset to prevent reuse.
    - Handles errors and flashes appropriate messages to the user.

```python
@app.route('/update_password_beneficiary', methods=['GET', 'POST'])
def update_password_beneficiary():
    if request.method == 'POST':
        email = request.form.get('email')
        new_password = request.form.get('new_password')

        # Connect to the database
        connection = db
        try:
            with connection.cursor() as cursor:
                # Update the beneficiary's password in the 'beneficiaries' table
                update_query = "UPDATE beneficiary SET password=%s WHERE email=%s"
                cursor.execute(update_query, (new_password, email))
                connection.commit()

                if cursor.rowcount == 0:
                    flash('No beneficiary found with this email address.', 'warning')
                else:
                    flash('Password updated successfully!', 'success')

        except Exception as e:
            flash('An error occurred while updating the password. Please try again.', 'danger')
        finally:
            connection.close()

        return redirect(url_for('beneficiary_login'))  # Redirect to login or another page as needed

    return render_template('update_password_beneficiary.html')  # Render the update password form

```

URL: /update_password_beneficiary

Methods:

- GET: To display the update password form.
- POST: To process the new password and update it in the database.

Functionality:

1. Handling the GET Request:
    - Renders the update_password_beneficiary.html template for users to input their new password.
2. Handling the POST Request:
    - Retrieves the email and new password from the form.
    - Establishes a database connection and updates the beneficiary's password based on the provided email.
    - Uses flash messages to inform the user about the success or failure of the update.

1.8 Other Pages With Routings

```
850   # Route for Learn More page
851   @app.route('/learnmore')
852   def learn_more():
853       return render_template('learnmore.html')
854
855   # Route for Contact page
856   @app.route('/contact')
857   def contact():
858       return render_template('contact.html')
859
860   # Route for About page
861   @app.route('/about')
862   def about():
863       return render_template('about.html')
864
865   # Blog Page route with Pie Chart
866   @app.route('/blog')
867   def blog():
868       return render_template('blog.html')
869
870   @app.route('/feedback_thank_you')
871   def feedback_thank_you():
872       return render_template('feedback_thank_you.html')
873
874   @app.route('/feedback')
875   def feedback():
876       return render_template('feedback.html')
877
```

/learnmore

- Method: GET
- Purpose: To provide additional information about the application or its features.
- Returns: Renders the learnmore.html template for users to read.

/contact

- Method: GET

- Purpose: To offer a means for users to contact the organization or support team.
- Returns: Renders the contact.html template for users to view contact details or a form.

/about

- Method: GET
- Purpose: To present information about the organization or project, including its mission and values.
- Returns: Renders the about.html template that describes the organization.

/blog

- Method: GET
- Purpose: To provide access to blog posts, potentially including analytics visualizations like a pie chart.
- Returns: Renders the blog.html template where users can read blog entries.

/feedback_thank_you

- Method: GET
- Purpose: To acknowledge receipt of user feedback.
- Returns: Renders the feedback_thank_you.html template to inform users that their feedback was received.

/feedback

- Method: GET
- Purpose: To allow users to submit their feedback about the application.
- Returns: Renders the feedback.html template containing a form for feedback submission.

1.8.1 Feedback Submission Routing

```
882  @app.route('/submit-feedback', methods=['POST'])
883  def submit_feedback():
884      # Get the form data
885      name = request.form.get('name')
886      rating = request.form.get('rating')
887      category = request.form.get('category')
888      feedback = request.form.get('feedback')
889
890      # Get current timestamp for 'created_at'
891      created_at = datetime.now().strftime('%Y-%m-%d %H:%M:%S')
892
893      # Assuming `db` is a connection object
894      conn = db  # Get the connection object
895      if conn is None:
896          return "Database connection error", 500
897
898      # Get a cursor from the connection
899      cursor = conn.cursor()
900
901      query = """
902          INSERT INTO feedback (name, rating, category, feedback, created_at)
903          VALUES (%s, %s, %s, %s, %s)
904      """
905      values = (name, rating, category, feedback, created_at)
906
907      try:
908          cursor.execute(query, values)
909          conn.commit()  # Commit the transaction
910      except Exception as e:
911          print(f"Error: {e}")
912          conn.rollback()  # Roll back if there's an error
913          return "An error occurred while submitting feedback", 500
914      finally:
915          # Close the cursor and connection after the operation
916          cursor.close()
917          conn.close()

918      return redirect(url_for('feedback_thank_you'))
919
```

URL :/submit-feedback

- **Method**: POST
- **Purpose**: To accept feedback submissions from users, including their name, rating, category, and comments.
- **Process**:
    - The route extracts form data from the incoming request.
    - It generates a timestamp for when the feedback was created.

- It attempts to insert the feedback into the feedback table in the database.
- If successful, it redirects the user to a thank-you page; if there's an error, it returns an error message.

- **Returns**:
  - If the feedback is successfully submitted, the user is redirected to the feedback_thank_you page.
  - If there's an error (e.g., database connection issues), it returns an error message with a 500 status code.

1.9 Logout Routing

```
545
546
547    # Logout route
548    @app.route('/logout')
549    def logout():
550        session.clear()
551        return redirect(url_for('home'))
```

URL :/logout

- **Method**: GET
- **Purpose**: To log the user out of the application by clearing the session data.
- **Process**:
  - The route clears all data stored in the session using session.clear(), effectively logging the user out.
  - After clearing the session, it redirects the user to the home page.
- **Returns**:
  - A redirect to the home page, indicating that the user has successfully logged out.

```
920
921 if __name__ == '__main__':
922     app.run(debug=True)
923
```

if __name__ == '__main__':

- This line checks whether the script is being run directly (as opposed to being imported as a module in another script).
- If the script is run directly, the code block under this condition will execute.

app.run(debug=True)

- This line starts the Flask development server.
- **debug=True**:
    - Enables debug mode, which allows for automatic reloading of the server when code changes are made.
    - Provides a debugger in the browser for easier error tracing during development. If an error occurs, it shows a detailed traceback, which can help in debugging.

CONCLUSION:

The **“Ahar Setu”** web application, developed using Flask, is a comprehensive platform that effectively connects food donors, primarily restaurants, with beneficiaries, including orphanages, shelters, and non-profit organizations that face food insecurity. Utilizing Python’s Flask framework allows for rapid development and easy scalability, while PyMySQL facilitates seamless database connectivity for managing user and food post data. The application includes user authentication features, allowing both donors and beneficiaries to register, log in, and manage their profiles securely. Beneficiaries can search for food donations by city, ensuring they find available resources quickly, while donors can effortlessly post details about surplus food items, complete with descriptions, quantities, and expiration dates. The system also integrates email functionalities through Flask-Mail, enabling notifications for password recovery

and updates on food reservations. This project not only addresses the pressing issue of food waste but also fosters community engagement by encouraging participation from local businesses and organizations. By creating a user-friendly interface and employing modern web technologies, the application aims to inspire a culture of sharing and compassion, ultimately contributing to a more sustainable food ecosystem where excess food is redirected to those in need.