Code Documentation

September 25 , 2024

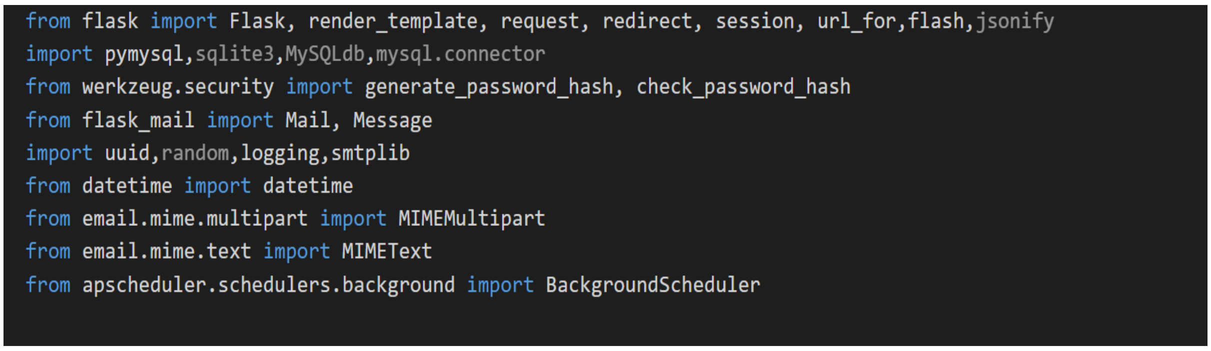
Flexforge Code Documentation

Introduction

This documentation provides an overview of the Flexforge web application, a platform designed to enhance the fitness journey for gym members and trainers. Flexforge offers tailored workout plans, personal training services, and nutrition guidance through a user-friendly interface, facilitating effective communication between users and trainers.

Within this documentation, you'll find details about the project's structure, the technologies used, and step-by-step instructions for setting up and navigating the application. This resource is intended to assist developers in understanding and working with the Flexforge codebase efficiently.

Libraries overview



```
from flask import Flask, render_template, request, redirect, session, url_for,flash,jsonify
import pymysql,sqlite3,MySQLdb,mysql.connector
from werkzeug.security import generate_password_hash, check_password_hash
from flask_mail import Mail, Message
import uuid,random,logging,smtplib
from datetime import datetime
from email.mime.multipart import MIMEMultipart
from email.mime.text import MIMEText
from apscheduler.schedulers.background import BackgroundScheduler
```

Code Overview

This section describes the main components of the code used in the Flexforge web application, which is built using the Flask framework. The application focuses on providing a comprehensive fitness platform for gym members and trainers.

Imports

- Flask: The core framework used to create the web application.

- render_template: Renders HTML templates for user interfaces.

- request, redirect, session, url_for, flash: Flask utilities for handling HTTP requests, redirections, session management, and user notifications.

- jsonify: Converts Python dictionaries into JSON responses for API interactions.

Database Connectivity

- pymysql, sqlite3, MySQLdb, mysql.connector: Libraries for connecting to various databases, allowing the application to store and retrieve user data, workout plans, and other essential information.

Security

- generate_password_hash, check_password_hash: Functions from `werkzeug.security` for securely hashing and verifying passwords, ensuring user data protection.

Email Functionality

- Flask-Mail: Used to manage email sending functionality, which may be utilized for user notifications, password resets, and communication between members and trainers.

- smtplib and email.mime: Standard libraries for composing and sending emails in a multi-part format.

Utilities

- uuid: Generates unique identifiers for users or sessions.

- random: Used for generating random values, possibly for features like selecting random workout plans.

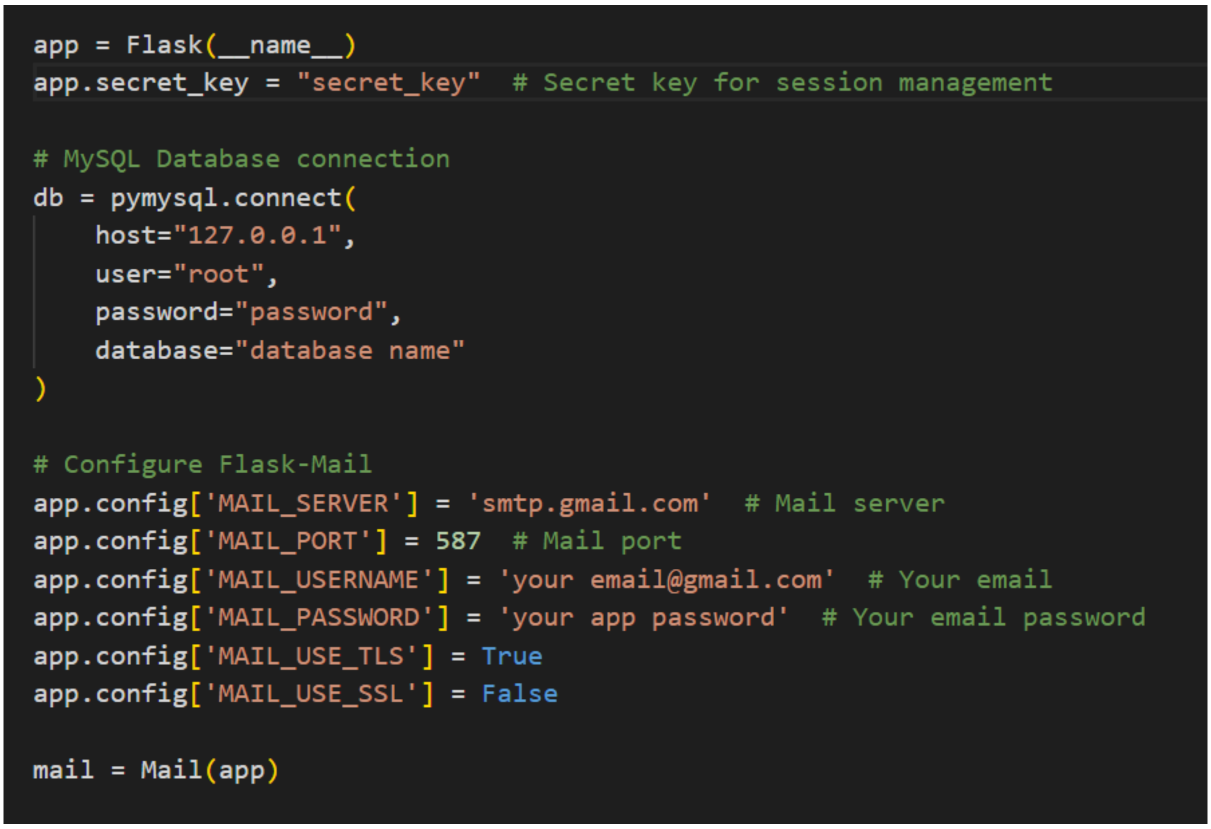
- logging: Implements logging for tracking application behavior and debugging purposes.

- datetime: Handles date and time-related operations, essential for tracking user activities and timestamps.

Scheduling

- BackgroundScheduler: From the `apscheduler` library, allows scheduling tasks to run in the background, which can be useful for regular updates or reminders for users.

Application Setup , Database Connection and Email Configuration



```python
app = Flask(__name__)
app.secret_key = "secret_key"  # Secret key for session management

# MySQL Database connection
db = pymysql.connect(
    host="127.0.0.1",
    user="root",
    password="password",
    database="database name"
)

# Configure Flask-Mail
app.config['MAIL_SERVER'] = 'smtp.gmail.com'  # Mail server
app.config['MAIL_PORT'] = 587  # Mail port
app.config['MAIL_USERNAME'] = 'your email@gmail.com'  # Your email
app.config['MAIL_PASSWORD'] = 'your app password'  # Your email password
app.config['MAIL_USE_TLS'] = True
app.config['MAIL_USE_SSL'] = False

mail = Mail(app)
```

This section outlines the initial setup of the Flask application and the configuration of email services.

Flask Application Initialization

- Creating the Flask App: The application is instantiated using `Flask(__name__)`, which creates an instance of the Flask class. This instance serves as the main entry point for the web application.

- Session Management: A secret key is set using `app.secret_key = "hello"`. This key is crucial for managing sessions securely, allowing for the storage of user-specific data across requests.

Database Connection

- MySQL Database Setup: The application connects to a MySQL database using the `pymysql` library. The connection parameters include:

  - host: The address of the database server (e.g., `127.0.0.1` for local development).

  - user: The username for the database (e.g., `root`).

  - password: The password for the specified user.

  - database: The name of the database to connect to (e.g., `mydb`).

Email Configuration

- Flask-Mail Integration: The Flask-Mail extension is configured to enable email functionality within the application. The configuration settings include:

  - MAIL_SERVER: Specifies the email server to use (e.g., `smtp.gmail.com` for Gmail).

  - MAIL_PORT: Defines the port for the email server (587 for TLS).

  - MAIL_USERNAME: Your email address used for sending emails.

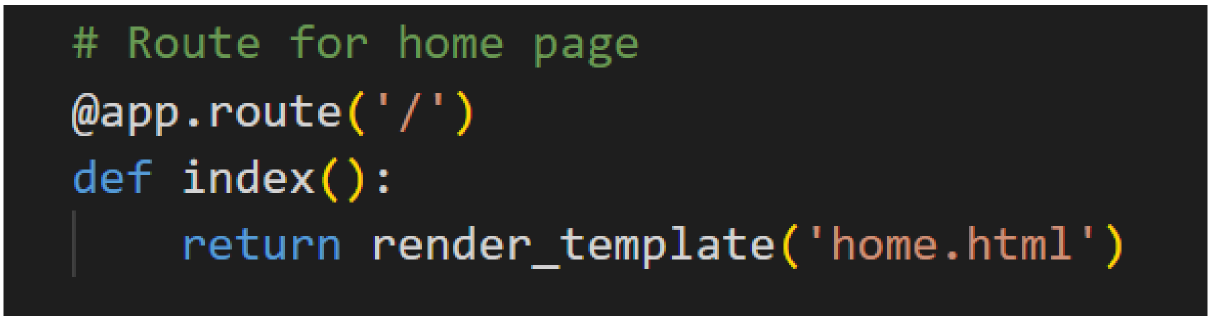
  - MAIL_PASSWORD: The password for your email account (note: consider using environment variables for security).

  - MAIL_USE_TLS: Enables TLS encryption for secure email sending (set to `True`).

  - MAIL_USE_SSL: Indicates whether to use SSL (set to `False`).

- Mail Instance: An instance of `Mail(app)` is created, enabling the application to send emails through the configured SMTP server.

Home Page Route



```python
# Route for home page
@app.route('/')
def index():
    return render_template('home.html')
```

Route Definition

- Endpoint: The route is defined using the `@app.route('/')` decorator, which specifies the home page of the application.

Functionality

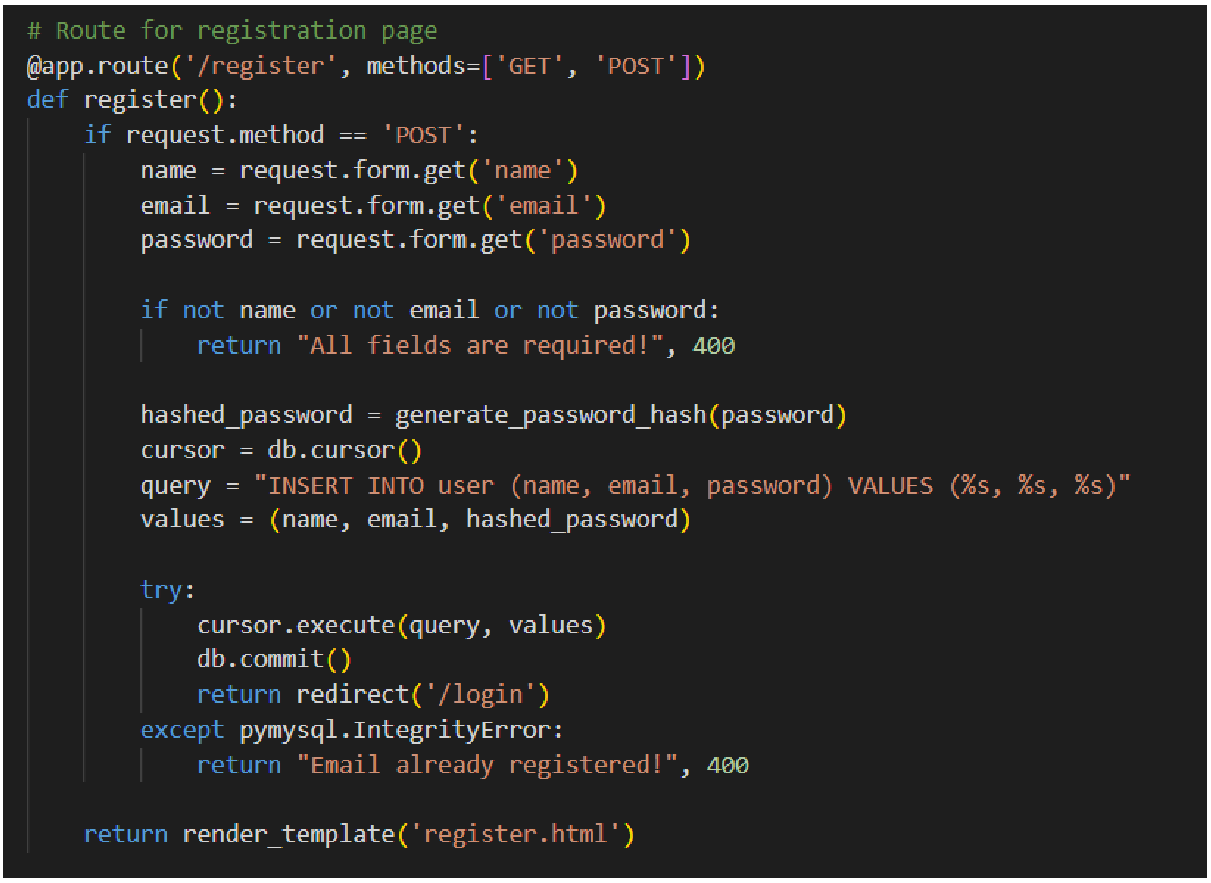
- Function Name: `index()`: This function is triggered when a user navigates to the home page.

- Rendering the Template: The function returns the rendered HTML template `home.html` using the `render_template()` function from Flask. This template serves as the main interface for users visiting the home page, displaying essential information about Flexforge and its offerings.

Purpose

The home page serves as the introductory point for users, providing them with an overview of the Flexforge platform and guiding them toward further interactions, such as signing up or logging in.

Member Registration Route



```python
# Route for registration page
@app.route('/register', methods=['GET', 'POST'])
def register():
    if request.method == 'POST':
        name = request.form.get('name')
        email = request.form.get('email')
        password = request.form.get('password')

        if not name or not email or not password:
            return "All fields are required!", 400

        hashed_password = generate_password_hash(password)
        cursor = db.cursor()
        query = "INSERT INTO user (name, email, password) VALUES (%s, %s, %s)"
        values = (name, email, hashed_password)

        try:
            cursor.execute(query, values)
            db.commit()
            return redirect('/login')
        except pymysql.IntegrityError:
            return "Email already registered!", 400

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

Route Definition

- Endpoint: The route is defined as `@app.route('/register', methods=['GET', 'POST'])`.

Functionality

- Form Rendering: If accessed via a `GET` request, it displays the registration form (`register.html`).

- Form Submission: Upon a `POST` request:
  - The form data (e.g., name, email, password) is captured.
  - The password is securely hashed using `generate_password_hash()` to ensure data security.

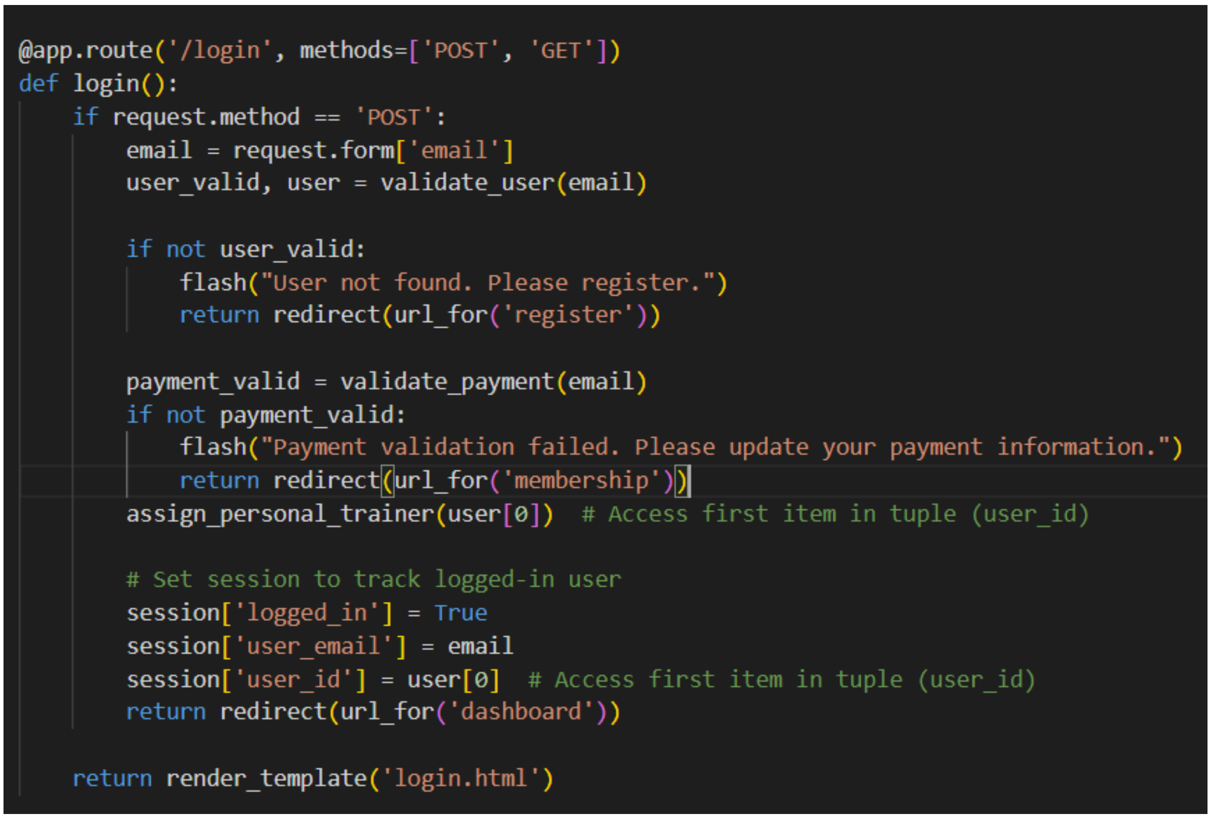
- Database Interaction: The user's details (name, email, hashed password) are stored in the `user` table in the MySQL database.

- Response: After successfully saving the data, the user is redirected to the appropriate page or shown a success message.

Purpose

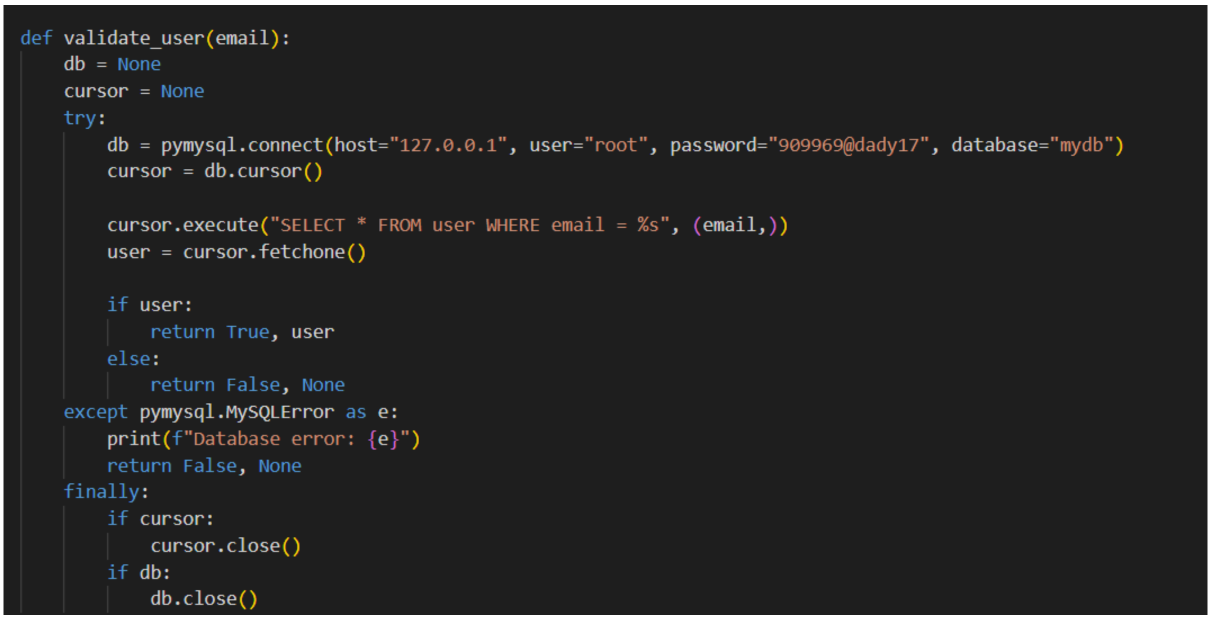
This route facilitates member registration by securely storing user information in the `user` table, allowing them access to the Flexforge platform.

Login Route



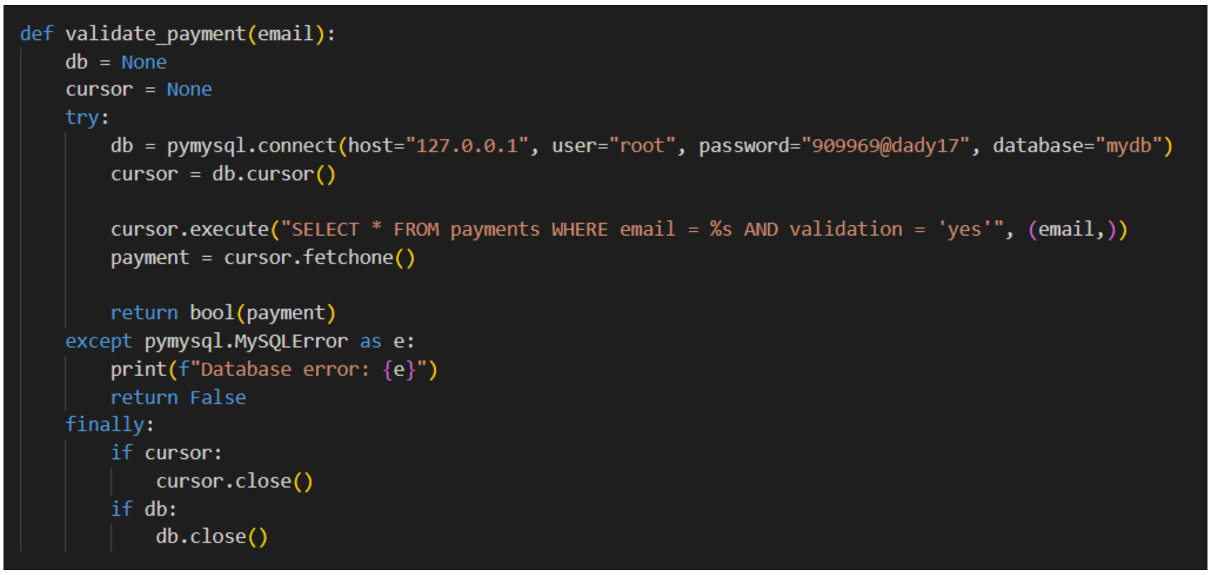
```
@app.route('/login', methods=['POST', 'GET'])
def login():
    if request.method == 'POST':
        email = request.form['email']
        user_valid, user = validate_user(email)

        if not user_valid:
            flash("User not found. Please register.")
            return redirect(url_for('register'))

        payment_valid = validate_payment(email)
        if not payment_valid:
            flash("Payment validation failed. Please update your payment information.")
            return redirect(url_for('membership'))
        assign_personal_trainer(user[0])  # Access first item in tuple (user_id)

        # Set session to track logged-in user
        session['logged_in'] = True
        session['user_email'] = email
        session['user_id'] = user[0]  # Access first item in tuple (user_id)
        return redirect(url_for('dashboard'))

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

Validate_user Function



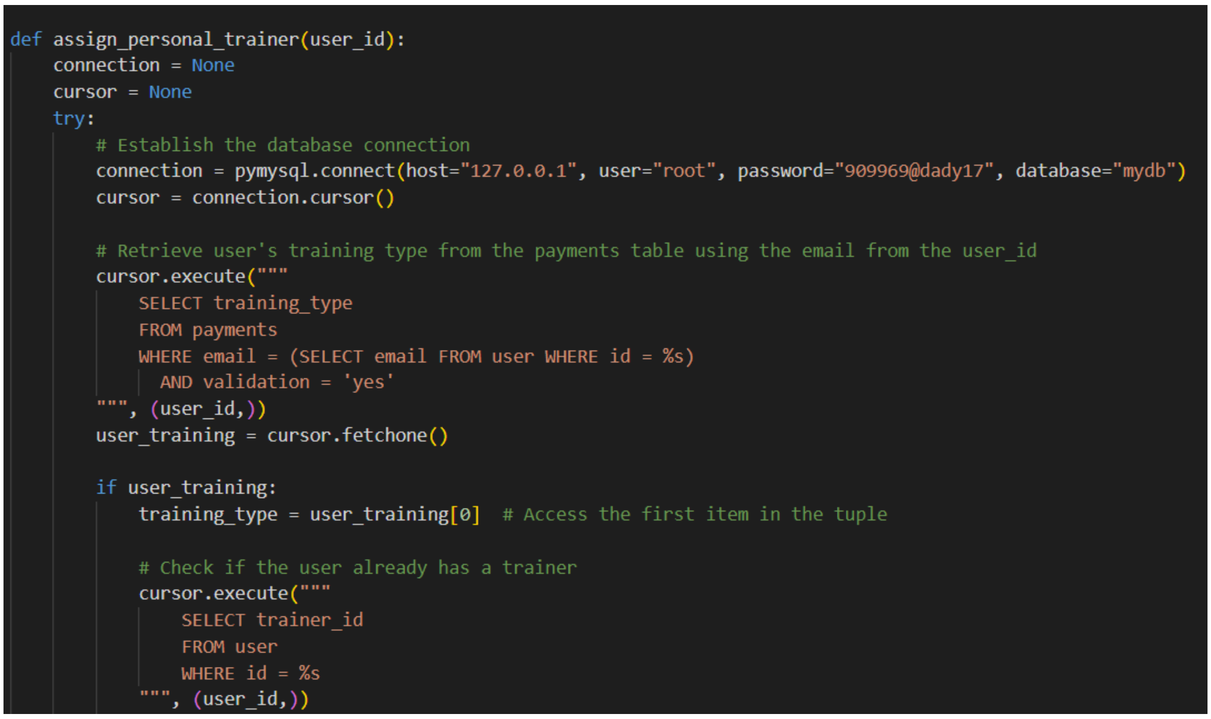
```
def validate_user(email):
    db = None
    cursor = None
    try:
        db = pymysql.connect(host="127.0.0.1", user="root", password="909969@dady17", database="mydb")
        cursor = db.cursor()

        cursor.execute("SELECT * FROM user WHERE email = %s", (email,))
        user = cursor.fetchone()

        if user:
            return True, user
        else:
            return False, None
    except pymysql.MySQLError as e:
        print(f"Database error: {e}")
        return False, None
    finally:
        if cursor:
            cursor.close()
        if db:
            db.close()
```

Validate_payment Function

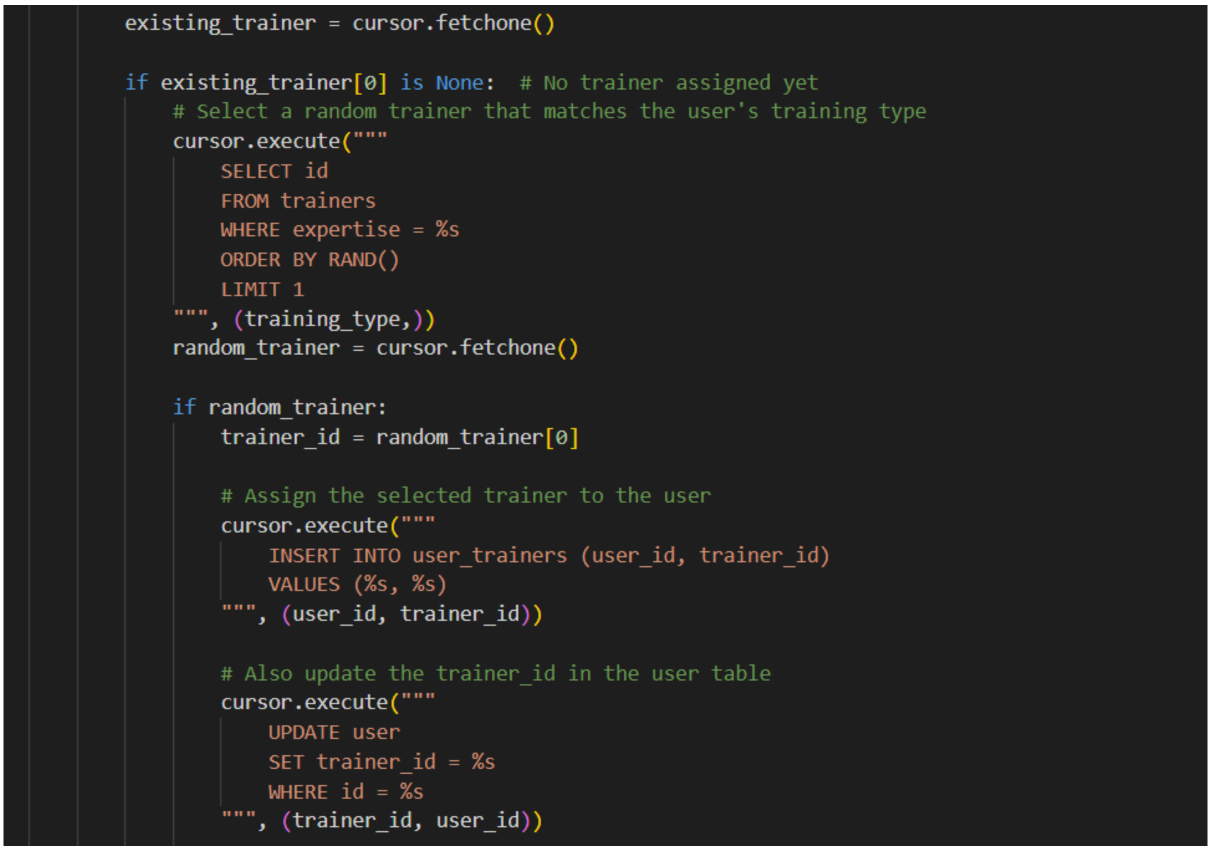


```
def validate_payment(email):
    db = None
    cursor = None
    try:
        db = pymysql.connect(host="127.0.0.1", user="root", password="909969@dady17", database="mydb")
        cursor = db.cursor()

        cursor.execute("SELECT * FROM payments WHERE email = %s AND validation = 'yes'", (email,))
        payment = cursor.fetchone()

        return bool(payment)
    except pymysql.MySQLError as e:
        print(f"Database error: {e}")
        return False
    finally:
        if cursor:
            cursor.close()
        if db:
            db.close()
```

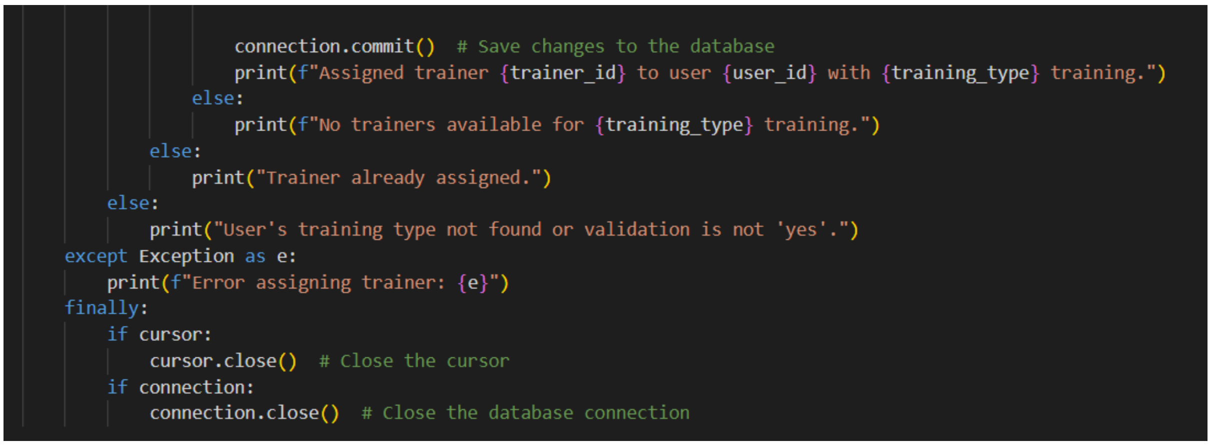
Assaign_personal_trainer Function



```python
def assign_personal_trainer(user_id):
    connection = None
    cursor = None
    try:
        # Establish the database connection
        connection = pymysql.connect(host="127.0.0.1", user="root", password="909969@dady17", database="mydb")
        cursor = connection.cursor()

        # Retrieve user's training type from the payments table using the email from the user_id
        cursor.execute("""
            SELECT training_type
            FROM payments
            WHERE email = (SELECT email FROM user WHERE id = %s)
              AND validation = 'yes'
        """, (user_id,))
        user_training = cursor.fetchone()

        if user_training:
            training_type = user_training[0]  # Access the first item in the tuple

            # Check if the user already has a trainer
            cursor.execute("""
                SELECT trainer_id
                FROM user
                WHERE id = %s
            """, (user_id,))
```



```python
            existing_trainer = cursor.fetchone()

            if existing_trainer[0] is None:  # No trainer assigned yet
                # Select a random trainer that matches the user's training type
                cursor.execute("""
                    SELECT id
                    FROM trainers
                    WHERE expertise = %s
                    ORDER BY RAND()
                    LIMIT 1
                """, (training_type,))
                random_trainer = cursor.fetchone()

                if random_trainer:
                    trainer_id = random_trainer[0]

                    # Assign the selected trainer to the user
                    cursor.execute("""
                        INSERT INTO user_trainers (user_id, trainer_id)
                        VALUES (%s, %s)
                    """, (user_id, trainer_id))

                    # Also update the trainer_id in the user table
                    cursor.execute("""
                        UPDATE user
                        SET trainer_id = %s
                        WHERE id = %s
                    """, (trainer_id, user_id))
```



```
                    connection.commit()  # Save changes to the database
                    print(f"Assigned trainer {trainer_id} to user {user_id} with {training_type} training.")
                else:
                    print(f"No trainers available for {training_type} training.")
            else:
                print("Trainer already assigned.")
        else:
            print("User's training type not found or validation is not 'yes'.")
    except Exception as e:
        print(f"Error assigning trainer: {e}")
    finally:
        if cursor:
            cursor.close()  # Close the cursor
        if connection:
            connection.close()  # Close the database connection
```

Route Definition

- Endpoint: This route is defined as `@app.route('/login', methods=['POST', 'GET'])`.

Functionality

- GET Request: Displays the login form (`login.html`).

- POST Request:

  - Captures the user's email from the form.

  - User Validation: The `validate_user(email)` function checks if the user exists. If not, the user is redirected to the registration page.

  - Payment Validation: The `validate_payment(email)` function verifies payment status. If invalid, the user is redirected to the membership page to update payment details.

  - Trainer Assignment: The `assign_personal_trainer(user_id)` function assigns a personal trainer to the user after successful login.

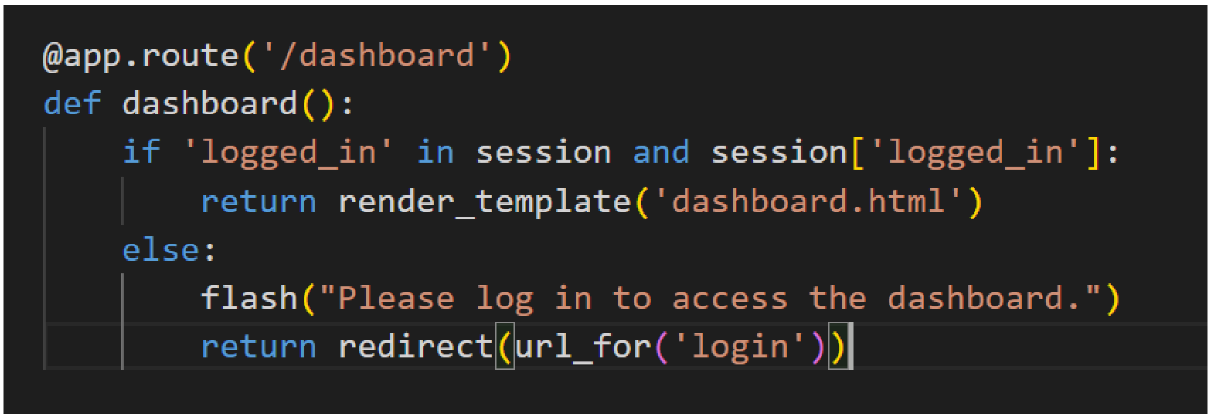
- Session Management: Upon successful login, the session is initialized to track the logged-in user by setting `session['logged_in']`, `session['user_email']`, and `session['user_id']`.

- Redirection: Once logged in, the user is redirected to the dashboard.

Purpose

This route enables user authentication, verifies payment, assigns a personal trainer, and manages user sessions for access to the Flexforge platform.

Member Dashboard Route


```
@app.route('/dashboard')
def dashboard():
    if 'logged_in' in session and session['logged_in']:
        return render_template('dashboard.html')
    else:
        flash("Please log in to access the dashboard.")
        return redirect(url_for('login'))
```

Route Definition

- **Endpoint**: The route is defined as @app.route('/dashboard').

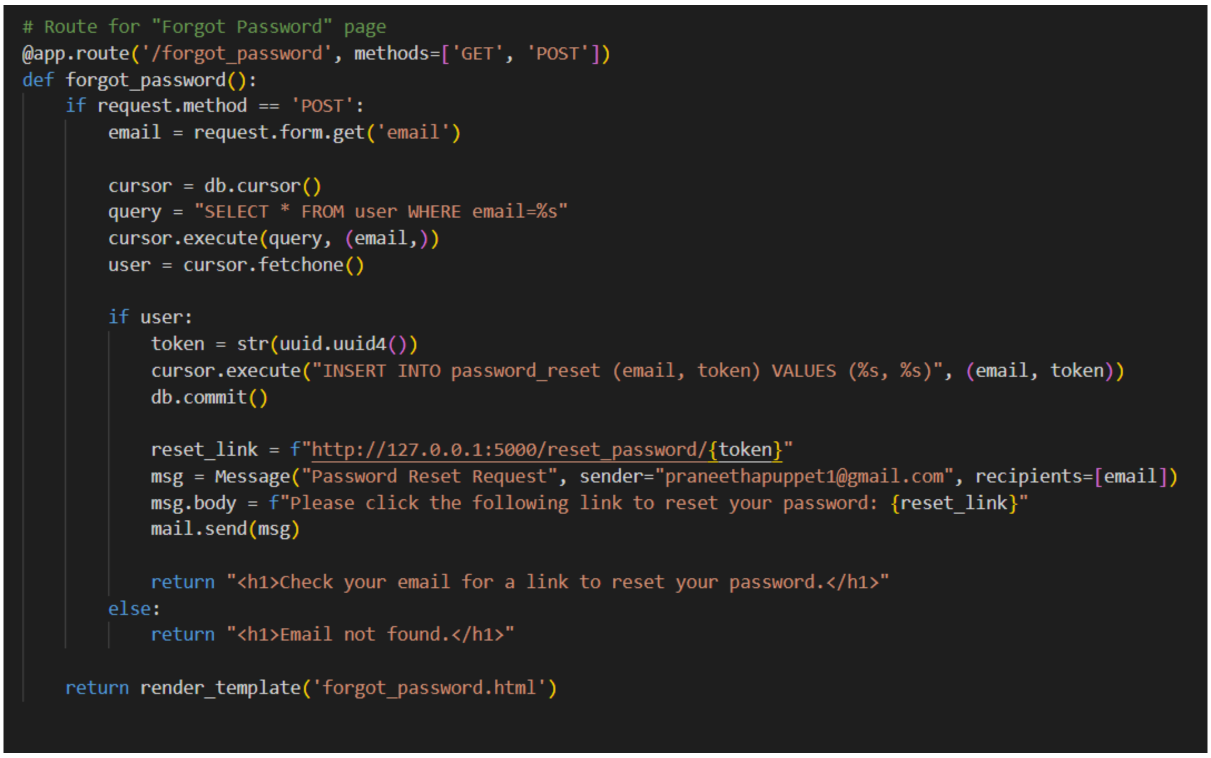
Functionality

- **Session Validation**:
    - The route checks if the user is logged in by confirming if 'logged_in' exists in the session and is set to True.
    - If the user is logged in, the dashboard.html template is rendered, providing access to their personalized dashboard.
- **Redirection for Unauthenticated Users**:
    - If the user is **not logged in**, a flash message is displayed: "Please log in to access the dashboard."
    - The user is then **redirected** to the login page (url_for('login')), ensuring only authenticated users can access the dashboard.

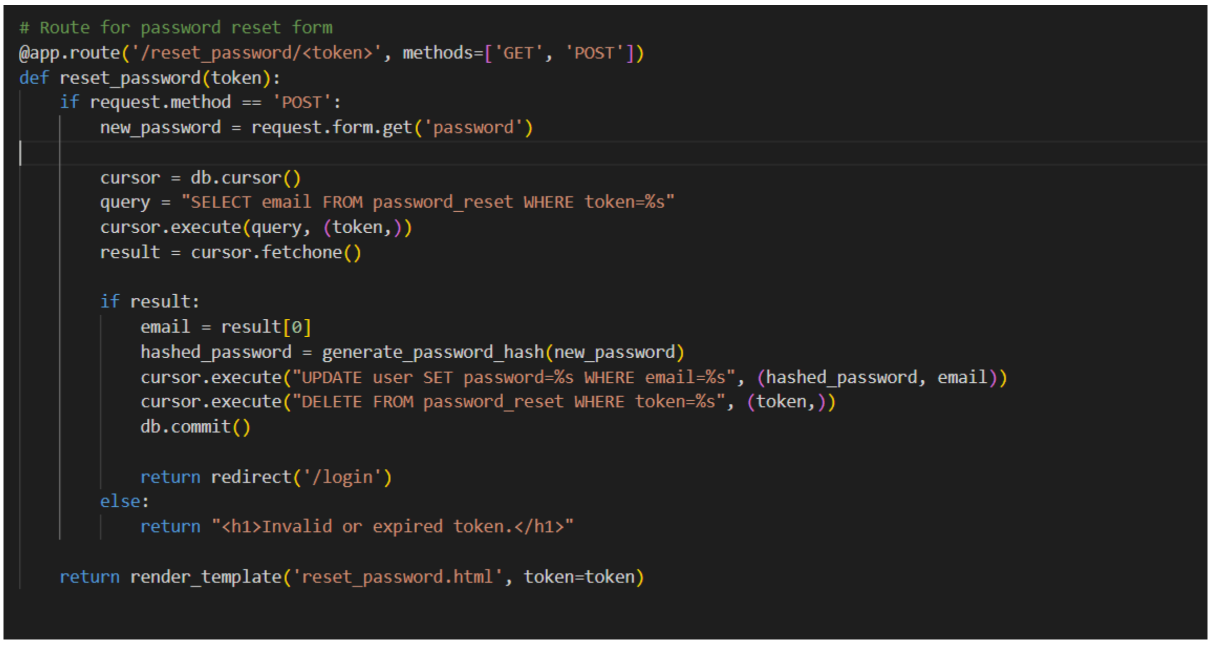
Purpose

This route enforces session-based authentication, ensuring that only logged-in users can access the member dashboard. Unauthenticated users are redirected to the login page.

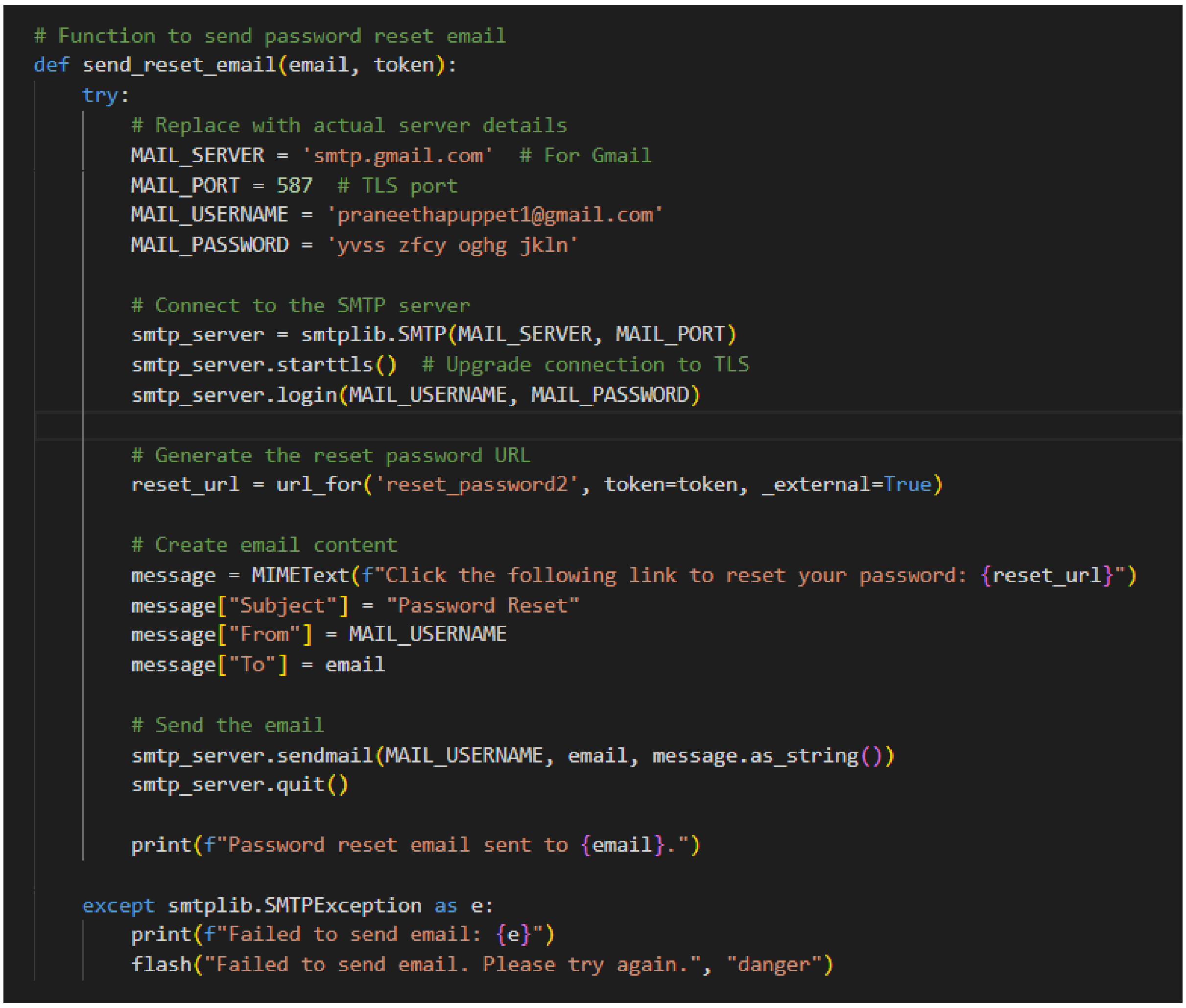
Forgot Password and Reset Password Routes



```python
# Route for "Forgot Password" page
@app.route('/forgot_password', methods=['GET', 'POST'])
def forgot_password():
    if request.method == 'POST':
        email = request.form.get('email')

        cursor = db.cursor()
        query = "SELECT * FROM user WHERE email=%s"
        cursor.execute(query, (email,))
        user = cursor.fetchone()

        if user:
            token = str(uuid.uuid4())
            cursor.execute("INSERT INTO password_reset (email, token) VALUES (%s, %s)", (email, token))
            db.commit()

            reset_link = f"http://127.0.0.1:5000/reset_password/{token}"
            msg = Message("Password Reset Request", sender="praneethapuppet1@gmail.com", recipients=[email])
            msg.body = f"Please click the following link to reset your password: {reset_link}"
            mail.send(msg)

            return "<h1>Check your email for a link to reset your password.</h1>"
        else:
            return "<h1>Email not found.</h1>"

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



```python
# Route for password reset form
@app.route('/reset_password/<token>', methods=['GET', 'POST'])
def reset_password(token):
    if request.method == 'POST':
        new_password = request.form.get('password')

        cursor = db.cursor()
        query = "SELECT email FROM password_reset WHERE token=%s"
        cursor.execute(query, (token,))
        result = cursor.fetchone()

        if result:
            email = result[0]
            hashed_password = generate_password_hash(new_password)
            cursor.execute("UPDATE user SET password=%s WHERE email=%s", (hashed_password, email))
            cursor.execute("DELETE FROM password_reset WHERE token=%s", (token,))
            db.commit()

            return redirect('/login')
        else:
            return "<h1>Invalid or expired token.</h1>"

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



```python
# Function to send password reset email
def send_reset_email(email, token):
    try:
        # Replace with actual server details
        MAIL_SERVER = 'smtp.gmail.com'  # For Gmail
        MAIL_PORT = 587  # TLS port
        MAIL_USERNAME = 'praneethapuppet1@gmail.com'
        MAIL_PASSWORD = 'yvss zfcy oghg jkln'

        # Connect to the SMTP server
        smtp_server = smtplib.SMTP(MAIL_SERVER, MAIL_PORT)
        smtp_server.starttls()  # Upgrade connection to TLS
        smtp_server.login(MAIL_USERNAME, MAIL_PASSWORD)

        # Generate the reset password URL
        reset_url = url_for('reset_password2', token=token, _external=True)

        # Create email content
        message = MIMEText(f"Click the following link to reset your password: {reset_url}")
        message["Subject"] = "Password Reset"
        message["From"] = MAIL_USERNAME
        message["To"] = email

        # Send the email
        smtp_server.sendmail(MAIL_USERNAME, email, message.as_string())
        smtp_server.quit()

        print(f"Password reset email sent to {email}.")

    except smtplib.SMTPException as e:
        print(f"Failed to send email: {e}")
        flash("Failed to send email. Please try again.", "danger")
```

1. Forgot Password Route

- **Endpoint**: @app.route('/forgot_password', methods=['GET', 'POST'])

Functionality:

- **POST Request**:
  - The user submits their email to request a password reset.
  - The code checks the user table in the database to see if the email exists.
  - If found, a unique token is generated using uuid.uuid4().
  - This token is inserted into the password_reset table along with the email.
  - A password reset email is sent to the user, containing a link to reset their password.
  - If the email is not found, a message is displayed indicating that the email is not registered.
- **GET Request**:
  - Renders the forgot_password.html form, allowing the user to input their email.

2. Password Reset Route

- **Endpoint**: @app.route('/reset_password/<token>', methods=['GET', 'POST'])

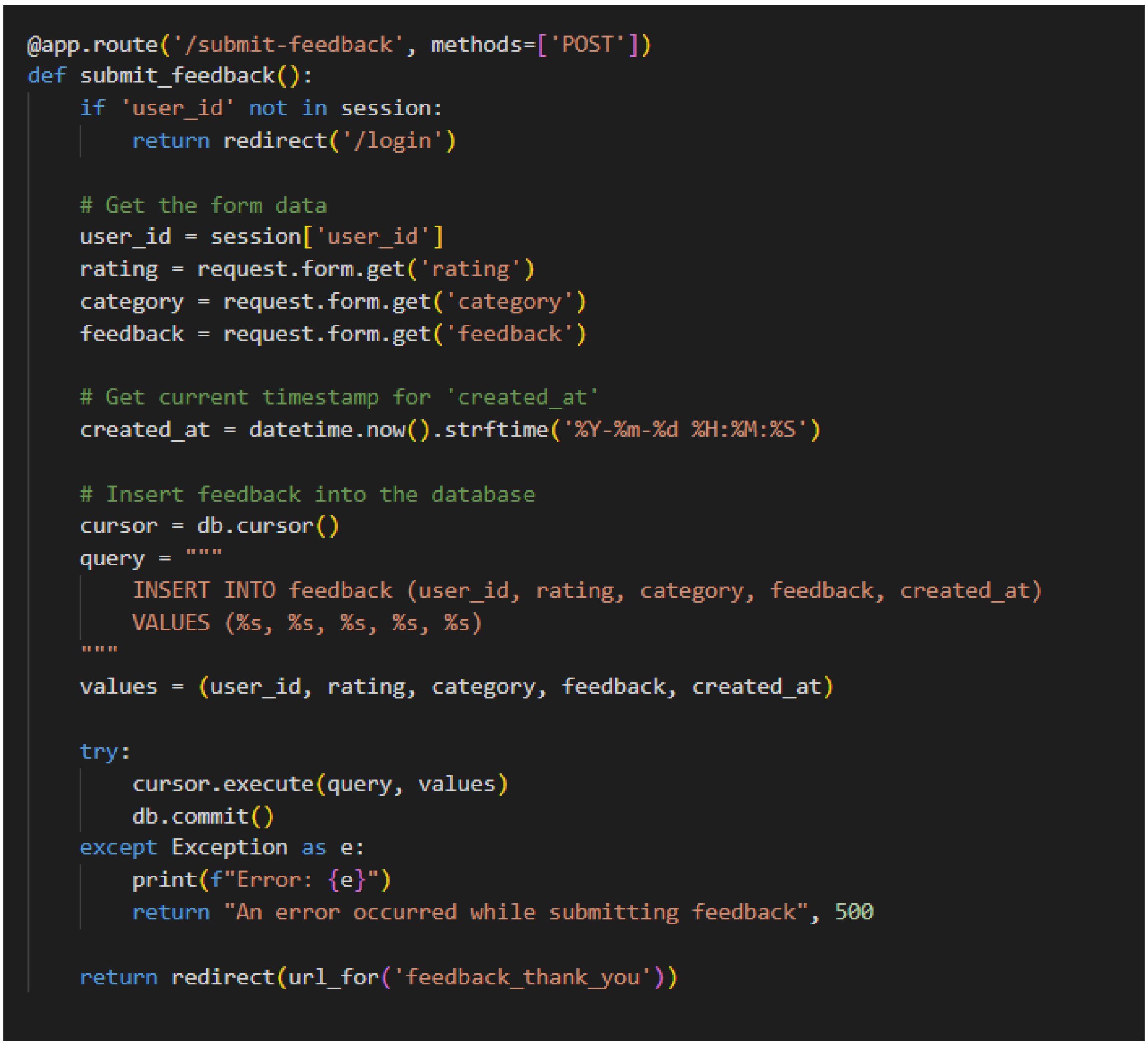
Functionality:

- **Token Validation**:
    - When the user clicks the reset link from their email, the token in the URL is used to verify their request.
    - The code checks the password_reset table to ensure the token is valid.
- **POST Request**:
    - The user enters a new password.
    - The password is hashed using generate_password_hash() and updated in the user table.
    - After successfully updating the password, the token is deleted from the password_reset table to ensure it cannot be reused.
- **GET Request**:
    - Renders the reset_password.html form for the user to input their new password.

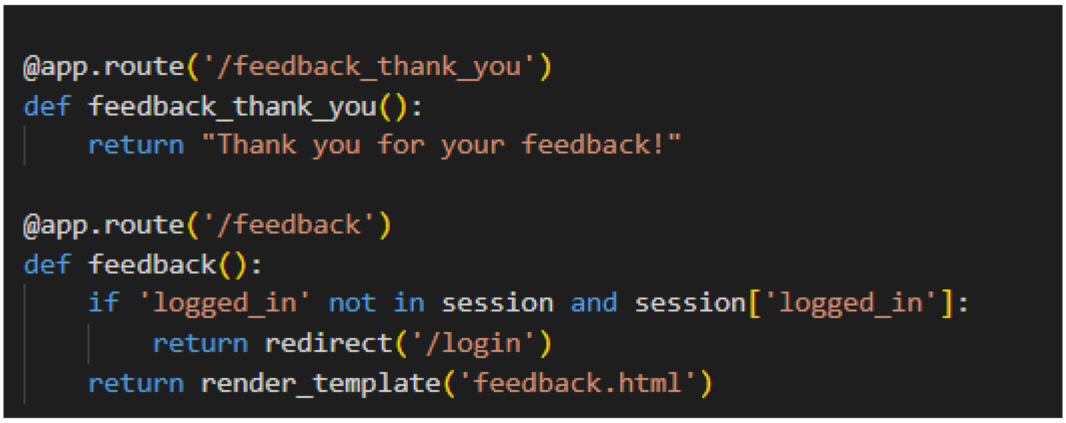
Key Aspects:

- **Security**: The password reset flow uses tokens to ensure secure and unique password reset links.
- **Email Integration**: Flask-Mail is used to send the reset link to the user's registered email.
- **Database Interaction**: Data is retrieved and updated in both user and password_reset tables to manage password reset requests securely.

Feedback Submission and Thank You Routes



```python
@app.route('/submit-feedback', methods=['POST'])
def submit_feedback():
    if 'user_id' not in session:
        return redirect('/login')

    # Get the form data
    user_id = session['user_id']
    rating = request.form.get('rating')
    category = request.form.get('category')
    feedback = request.form.get('feedback')

    # Get current timestamp for 'created_at'
    created_at = datetime.now().strftime('%Y-%m-%d %H:%M:%S')

    # Insert feedback into the database
    cursor = db.cursor()
    query = """
        INSERT INTO feedback (user_id, rating, category, feedback, created_at)
        VALUES (%s, %s, %s, %s, %s)
    """
    values = (user_id, rating, category, feedback, created_at)

    try:
        cursor.execute(query, values)
        db.commit()
    except Exception as e:
        print(f"Error: {e}")
        return "An error occurred while submitting feedback", 500

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



```python
@app.route('/feedback_thank_you')
def feedback_thank_you():
    return "Thank you for your feedback!"

@app.route('/feedback')
def feedback():
    if 'logged_in' not in session and session['logged_in']:
        return redirect('/login')
    return render_template('feedback.html')
```

1. Feedback Route

- **Endpoint**: @app.route('/feedback')

Functionality:

- This route displays the feedback form for logged-in users.
- It checks if the user is logged in by verifying the session. If not logged in, the user is redirected to the login page.
- If the user is logged in, it renders the feedback.html template, where the user can submit feedback.

2. Submit Feedback Route

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

Functionality:

- This route handles the form submission for feedback.
- The session is checked to ensure the user is logged in.
- Form data including rating, category, and feedback is retrieved from the user's input.
- The current timestamp is generated using datetime.now() to record when the feedback was submitted.
- The feedback is then inserted into the feedback table in the database with the fields: user_id, rating, category, feedback, and created_at.
- After successful submission, the user is redirected to the "Thank You" page (/feedback_thank_you).

3. Feedback Thank You Route

- **Endpoint**: @app.route('/feedback_thank_you')

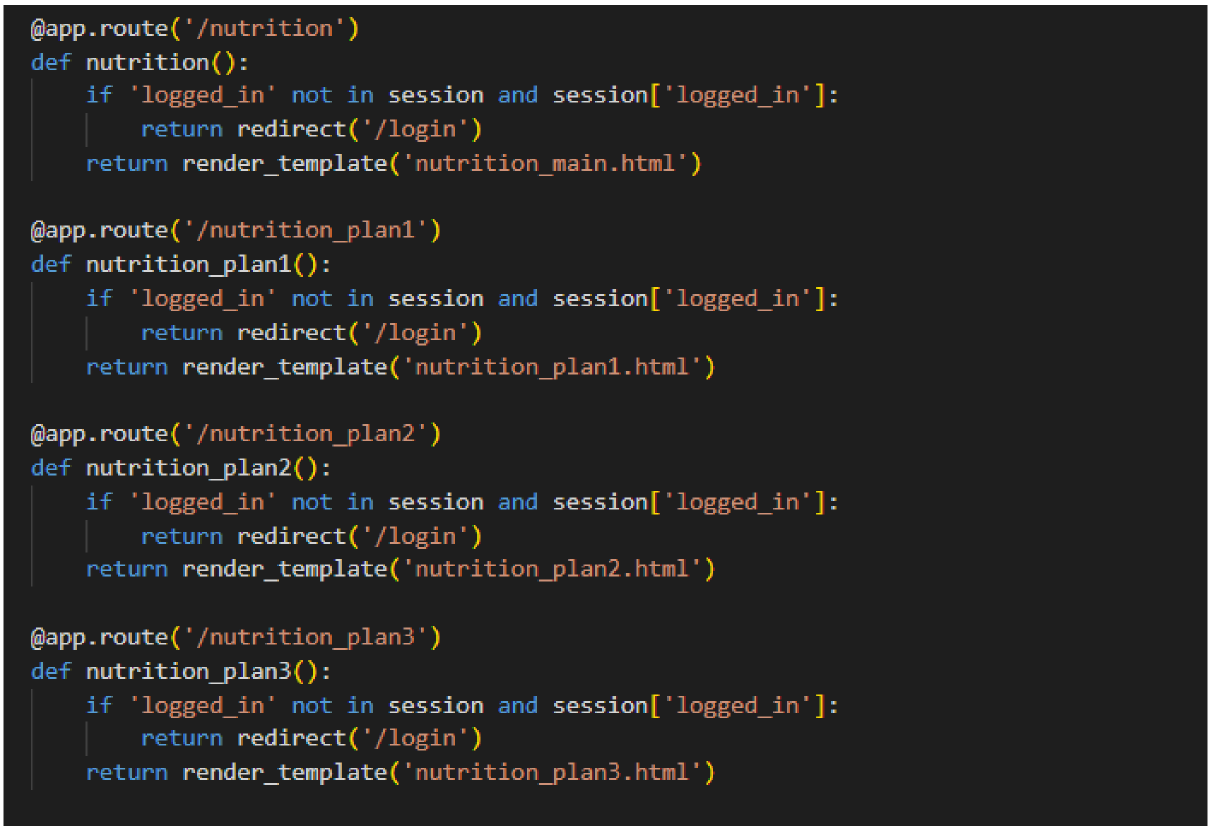
Functionality:

- After successfully submitting feedback, the user is redirected to this route.
- It displays a simple thank you message, acknowledging the feedback submission.

Key Aspects:

- **Database Interaction**: Feedback details are stored in the feedback table, and the created_at timestamp is used to track when the feedback was submitted.
- **Session Management**: Users must be logged in to access the feedback form and submit feedback.
- **Form Validation**: Data is captured from the feedback form and validated server-side before being stored in the database.

Nutrition Main Route



```python
@app.route('/nutrition')
def nutrition():
    if 'logged_in' not in session and session['logged_in']:
        return redirect('/login')
    return render_template('nutrition_main.html')

@app.route('/nutrition_plan1')
def nutrition_plan1():
    if 'logged_in' not in session and session['logged_in']:
        return redirect('/login')
    return render_template('nutrition_plan1.html')

@app.route('/nutrition_plan2')
def nutrition_plan2():
    if 'logged_in' not in session and session['logged_in']:
        return redirect('/login')
    return render_template('nutrition_plan2.html')

@app.route('/nutrition_plan3')
def nutrition_plan3():
    if 'logged_in' not in session and session['logged_in']:
        return redirect('/login')
    return render_template('nutrition_plan3.html')
```

- **Endpoint**: @app.route('/nutrition')

Functionality:

- This route displays the main nutrition page (nutrition_main.html).
- The session is checked to confirm the user is logged in; otherwise, the user is redirected to the login page.

2. Individual Nutrition Plan Routes

- **Endpoints**:
    - /nutrition_plan1
    - /nutrition_plan2
    - /nutrition_plan3
    - /nutrition_plan4
    - /nutrition_plan5
    - /nutrition_plan6
    - /nutrition_plan7

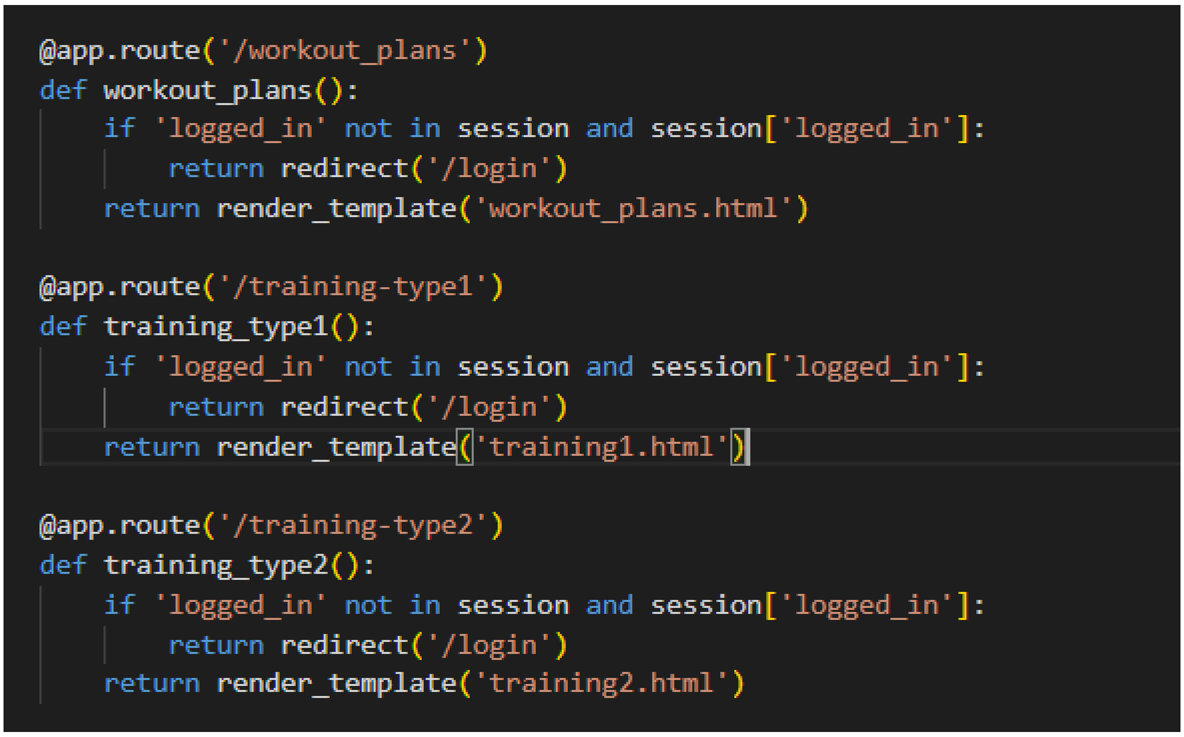
Functionality:

- Each route corresponds to a specific nutrition plan page (e.g., nutrition_plan1.html, nutrition_plan2.html, etc.).
- Similar to the main nutrition page, the session is checked to ensure the user is logged in.
- If the user is not logged in, they are redirected to the login page.

Key Aspects:

- **Session Management**: Users must be logged in to view any of the nutrition plans.
- **Navigation**: The routes serve different HTML templates, allowing users to access various nutrition plans once logged in.
- **User Restriction**: By implementing session checks, unauthorized access to nutrition plans is prevented.

Workout Plans Route



```python
@app.route('/workout_plans')
def workout_plans():
    if 'logged_in' not in session and session['logged_in']:
        return redirect('/login')
    return render_template('workout_plans.html')

@app.route('/training-type1')
def training_type1():
    if 'logged_in' not in session and session['logged_in']:
        return redirect('/login')
    return render_template('training1.html')

@app.route('/training-type2')
def training_type2():
    if 'logged_in' not in session and session['logged_in']:
        return redirect('/login')
    return render_template('training2.html')
```

- **Endpoint**: @app.route('/workout_plans')

Functionality:

- This route serves the main workout plans page (workout_plans.html).

- Before rendering the page, the session is checked to ensure the user is logged in. If not, the user is redirected to the login page (/login).

2. Training Type Routes

- **Endpoints**:
    - /training-type1
    - /training-type2
    - /training-type3
    - /training-type4
    - /training-type5
    - /training-type6
    - /training-type7

Functionality:

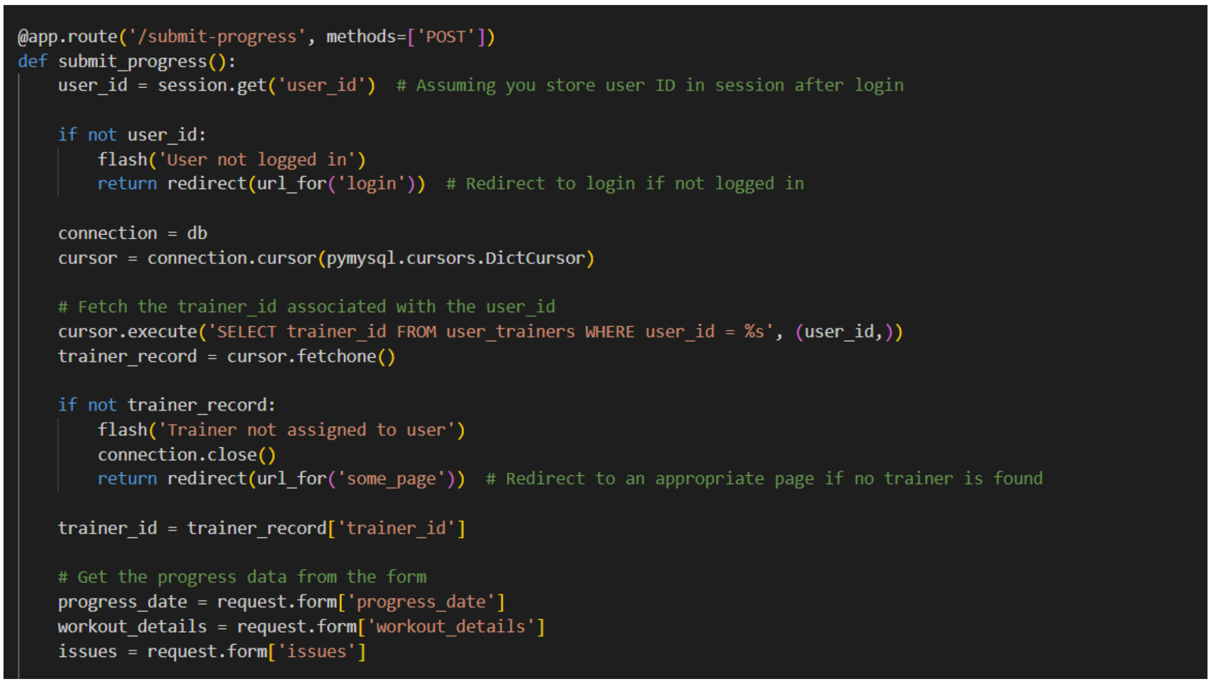
- These routes correspond to two different training types, each rendering a separate HTML page (training1.html and training2.html).
- Similar to the workout plans, the session is checked to verify that the user is logged in before the page is rendered.
- If the user is not logged in, they are redirected to the login page.

Key Aspects:

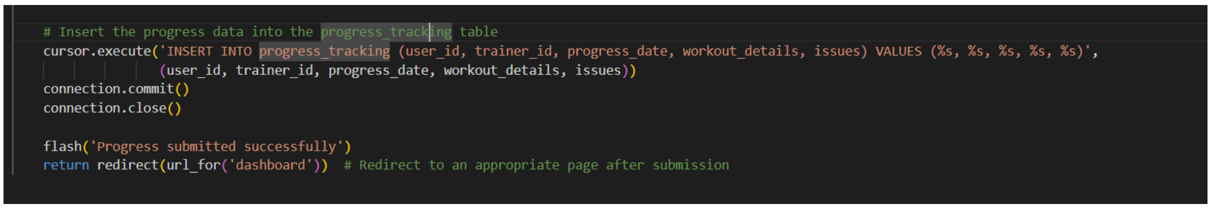
- **Session Management**: Every route ensures that the user must be logged in to access the workout plans or training type pages.
- **Customization**: These routes allow the user to navigate between different types of training programs, providing flexibility in accessing various workout resources.

Note: Multiple routes have been added to handle the training and nutrition subsections of the platform. Each route corresponds to specific pages, such as different training types or nutrition plans, ensuring organized navigation. User authentication is enforced to restrict access to these subroutes, enhancing security and user experience.

Overview of the Submit Progress Route



```python
@app.route('/submit-progress', methods=['POST'])
def submit_progress():
    user_id = session.get('user_id')  # Assuming you store user ID in session after login

    if not user_id:
        flash('User not logged in')
        return redirect(url_for('login'))  # Redirect to login if not logged in

    connection = db
    cursor = connection.cursor(pymysql.cursors.DictCursor)

    # Fetch the trainer_id associated with the user_id
    cursor.execute('SELECT trainer_id FROM user_trainers WHERE user_id = %s', (user_id,))
    trainer_record = cursor.fetchone()

    if not trainer_record:
        flash('Trainer not assigned to user')
        connection.close()
        return redirect(url_for('some_page'))  # Redirect to an appropriate page if no trainer is found

    trainer_id = trainer_record['trainer_id']

    # Get the progress data from the form
    progress_date = request.form['progress_date']
    workout_details = request.form['workout_details']
    issues = request.form['issues']
```



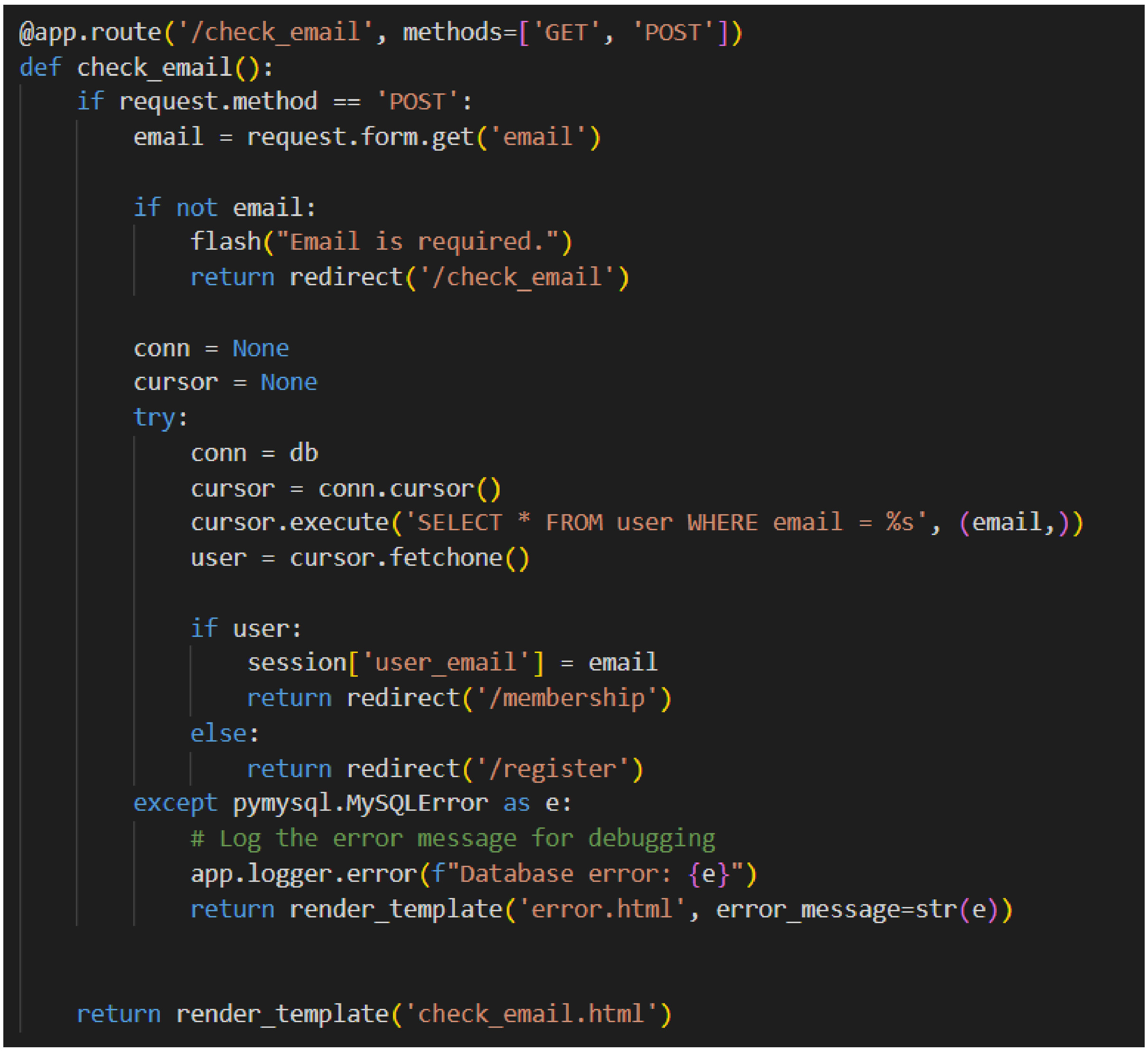
```python
    # Insert the progress data into the progress_tracking table
    cursor.execute('INSERT INTO progress_tracking (user_id, trainer_id, progress_date, workout_details, issues) VALUES (%s, %s, %s, %s, %s)',
                   (user_id, trainer_id, progress_date, workout_details, issues))
    connection.commit()
    connection.close()

    flash('Progress submitted successfully')
    return redirect(url_for('dashboard'))  # Redirect to an appropriate page after submission
```

This route manages the submission of workout progress from logged-in users. It ensures that progress is recorded alongside the assigned trainer's information, enhancing the tracking of individual user workouts.

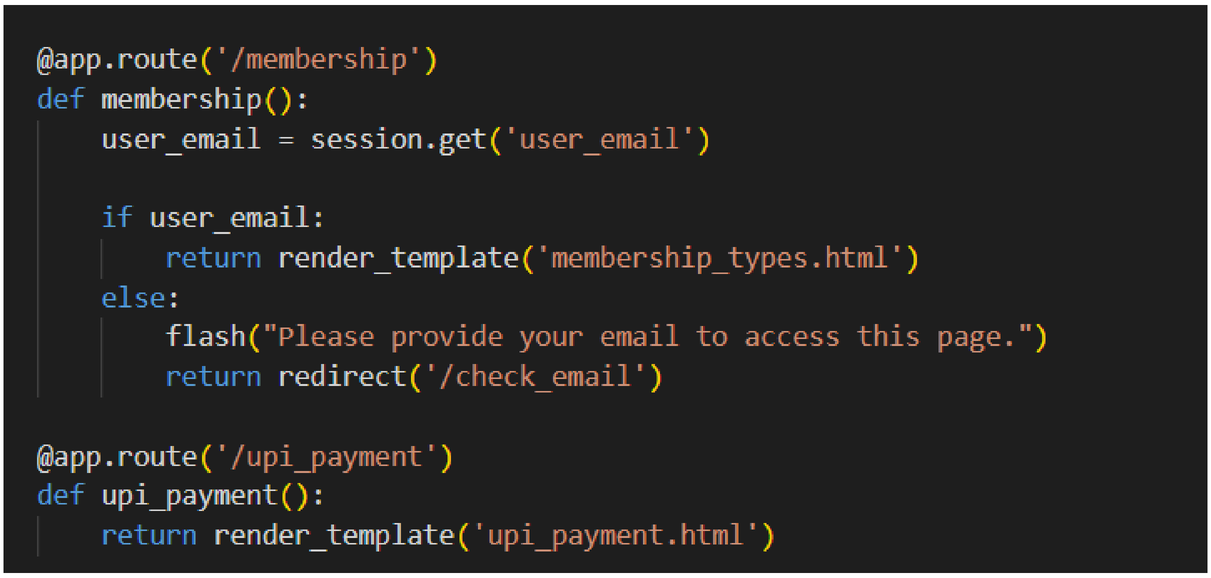
1. **User Authentication**:
   - Verifies if the user is logged in by checking the session for a user_id. If not logged in, it redirects to the login page.
2. **Trainer Verification**:
   - Retrieves the trainer_id associated with the user from the user_trainers table. If no trainer is found, it notifies the user and redirects them accordingly.
3. **Form Data Collection**:
   - Gathers the progress details from the submitted form, including:
     - progress_date: Date of the progress update.
     - workout_details: Description of the workout performed.
     - issues: Any challenges faced during the workout.

4. **Database Insertion**:
   - Inserts the progress data into the progress_tracking table, linking it to both the user and their assigned trainer.
5. **User Feedback**:
   - Upon successful submission, the user is informed of the success and redirected to their member dashboard.

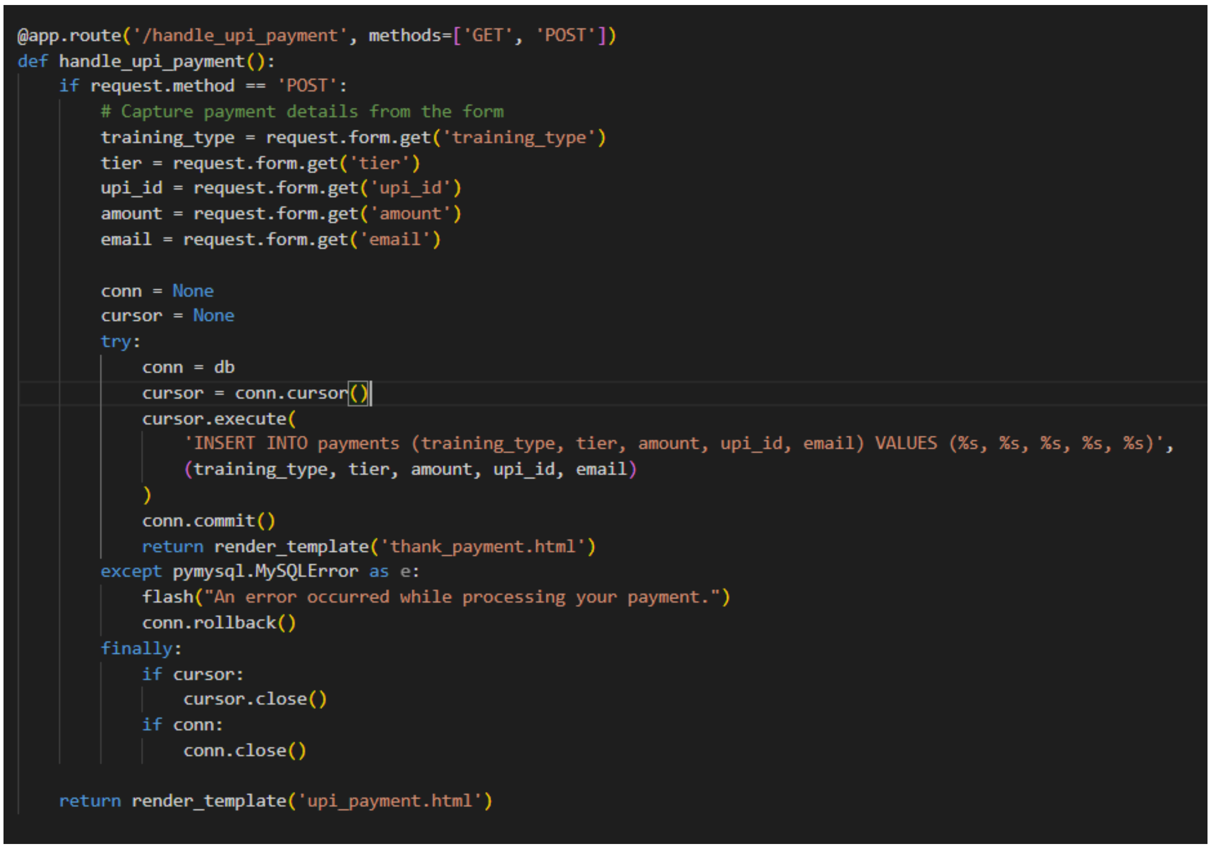
Overview of the Email Check and Payment Handling Routes



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

        if not email:
            flash("Email is required.")
            return redirect('/check_email')

        conn = None
        cursor = None
        try:
            conn = db
            cursor = conn.cursor()
            cursor.execute('SELECT * FROM user WHERE email = %s', (email,))
            user = cursor.fetchone()

            if user:
                session['user_email'] = email
                return redirect('/membership')
            else:
                return redirect('/register')
        except pymysql.MySQLError as e:
            # Log the error message for debugging
            app.logger.error(f"Database error: {e}")
            return render_template('error.html', error_message=str(e))


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



```python
@app.route('/membership')
def membership():
    user_email = session.get('user_email')

    if user_email:
        return render_template('membership_types.html')
    else:
        flash("Please provide your email to access this page.")
        return redirect('/check_email')

@app.route('/upi_payment')
def upi_payment():
    return render_template('upi_payment.html')
```



```python
@app.route('/handle_upi_payment', methods=['GET', 'POST'])
def handle_upi_payment():
    if request.method == 'POST':
        # Capture payment details from the form
        training_type = request.form.get('training_type')
        tier = request.form.get('tier')
        upi_id = request.form.get('upi_id')
        amount = request.form.get('amount')
        email = request.form.get('email')

        conn = None
        cursor = None
        try:
            conn = db
            cursor = conn.cursor()
            cursor.execute(
                'INSERT INTO payments (training_type, tier, amount, upi_id, email) VALUES (%s, %s, %s, %s, %s)',
                (training_type, tier, amount, upi_id, email)
            )
            conn.commit()
            return render_template('thank_payment.html')
        except pymysql.MySQLError as e:
            flash("An error occurred while processing your payment.")
            conn.rollback()
        finally:
            if cursor:
                cursor.close()
            if conn:
                conn.close()

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

1. Email Check Route (/check_email)

- **Purpose**: This route verifies if the email provided by the user is associated with an existing account.
- **Functionality**:
    - **POST Request Handling**: Upon form submission, it checks if the email field is filled. If empty, it flashes an error message and redirects back to the email check page.

- **Database Query**: It queries the user table for the provided email. If found, it saves the email in the session and redirects to the membership page; if not, it redirects to the registration page.
- **Error Handling**: It captures any database errors and logs them, rendering an error page if an exception occurs.

2. Membership Route (/membership)

- **Purpose**: This route displays membership options to users who have provided their email.
- **Functionality**:
    - It checks if the user's email is stored in the session. If present, it renders the membership types page; if not, it flashes a message and redirects to the email check page.

3. UPI Payment Route (/upi_payment)

- **Purpose**: This route renders the UPI payment form for users.
- **Functionality**:
    - Simply serves the upi_payment.html template for users to fill in their payment details.

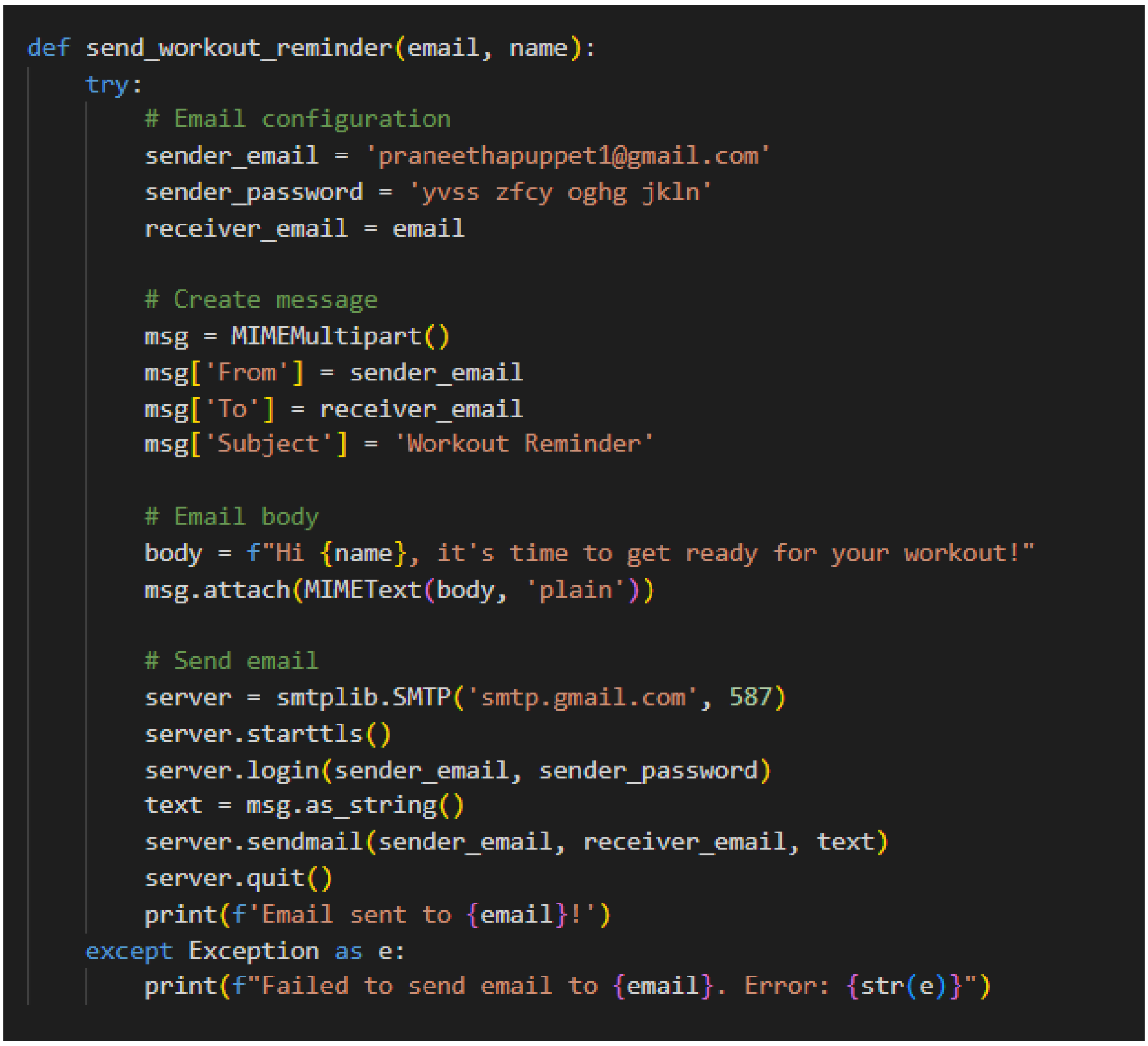
4. Handle UPI Payment Route (/handle_upi_payment)

- **Purpose**: This route processes the UPI payment details submitted by the user.
- **Functionality**:
    - **POST Request Handling**: It captures training type, membership tier, UPI ID, amount, and email from the form.
    - **Database Insertion**: It inserts these details into the payments table.
    - **Error Handling**: If a database error occurs during the insertion, it rolls back the transaction and flashes an error message. It also ensures proper closure of the database cursor and connection.
    - On successful payment processing, it renders a thank you page (thank_payment.html).

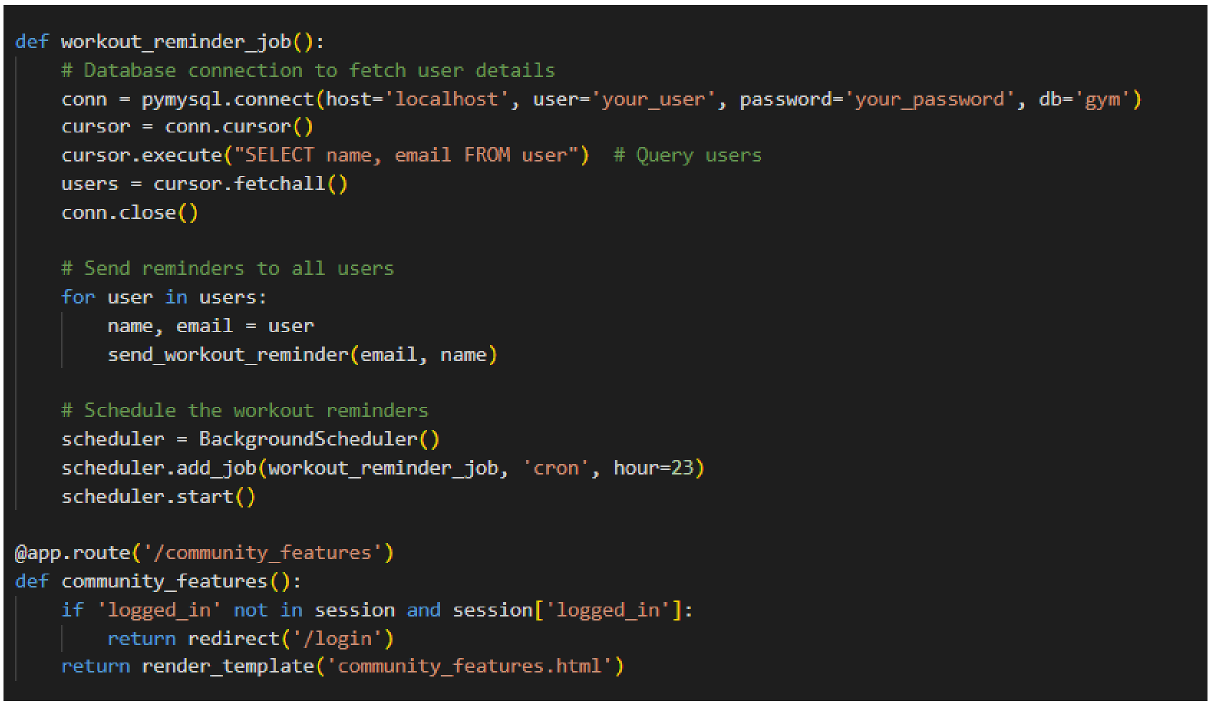
Summary

These routes form a crucial part of the application, facilitating user registration, membership selection, and payment processing while ensuring robust error handling and session management for a smooth user experience.

Overview of the Workout Reminder Functionality



```python
def send_workout_reminder(email, name):
    try:
        # Email configuration
        sender_email = 'praneethapuppet1@gmail.com'
        sender_password = 'yvss zfcy oghg jkln'
        receiver_email = email

        # Create message
        msg = MIMEMultipart()
        msg['From'] = sender_email
        msg['To'] = receiver_email
        msg['Subject'] = 'Workout Reminder'

        # Email body
        body = f"Hi {name}, it's time to get ready for your workout!"
        msg.attach(MIMEText(body, 'plain'))

        # Send email
        server = smtplib.SMTP('smtp.gmail.com', 587)
        server.starttls()
        server.login(sender_email, sender_password)
        text = msg.as_string()
        server.sendmail(sender_email, receiver_email, text)
        server.quit()
        print(f'Email sent to {email}!')
    except Exception as e:
        print(f"Failed to send email to {email}. Error: {str(e)}")
```



```python
def workout_reminder_job():
    # Database connection to fetch user details
    conn = pymysql.connect(host='localhost', user='your_user', password='your_password', db='gym')
    cursor = conn.cursor()
    cursor.execute("SELECT name, email FROM user")  # Query users
    users = cursor.fetchall()
    conn.close()

    # Send reminders to all users
    for user in users:
        name, email = user
        send_workout_reminder(email, name)

    # Schedule the workout reminders
    scheduler = BackgroundScheduler()
    scheduler.add_job(workout_reminder_job, 'cron', hour=23)
    scheduler.start()

@app.route('/community_features')
def community_features():
    if 'logged_in' not in session and session['logged_in']:
        return redirect('/login')
    return render_template('community_features.html')
```

1. Send Workout Reminder Function (send_workout_reminder)

- **Purpose**: This function sends a workout reminder email to users.
- **Parameters**:
  - email: The recipient's email address.
  - name: The recipient's name for personalization in the email.
- **Functionality**:
  - **Email Configuration**: Sets up the sender's email and password for authentication. (Make sure to handle passwords securely, avoiding hardcoding.)
  - **Message Creation**: Uses the MIMEMultipart class to create an email message with a subject and a personalized body.
  - **Sending Email**: Connects to Gmail's SMTP server, starts TLS for security, logs in with the sender's credentials, and sends the email.
  - **Error Handling**: Catches exceptions during the email sending process and prints an error message if it fails.

2. Workout Reminder Job Function (workout_reminder_job)

- **Purpose**: This function retrieves user details from the database and sends workout reminders.
- **Functionality**:
  - **Database Connection**: Connects to the MySQL database to fetch user names and email addresses.

- **Query Execution**: Executes a SQL query to select the name and email columns from the user table.
- **Loop Through Users**: Iterates over the retrieved user records and calls the send_workout_reminder function for each user to send an email.
- **Scheduler**: Uses a BackgroundScheduler to schedule the reminder job to run daily at a specified hour (23:00).

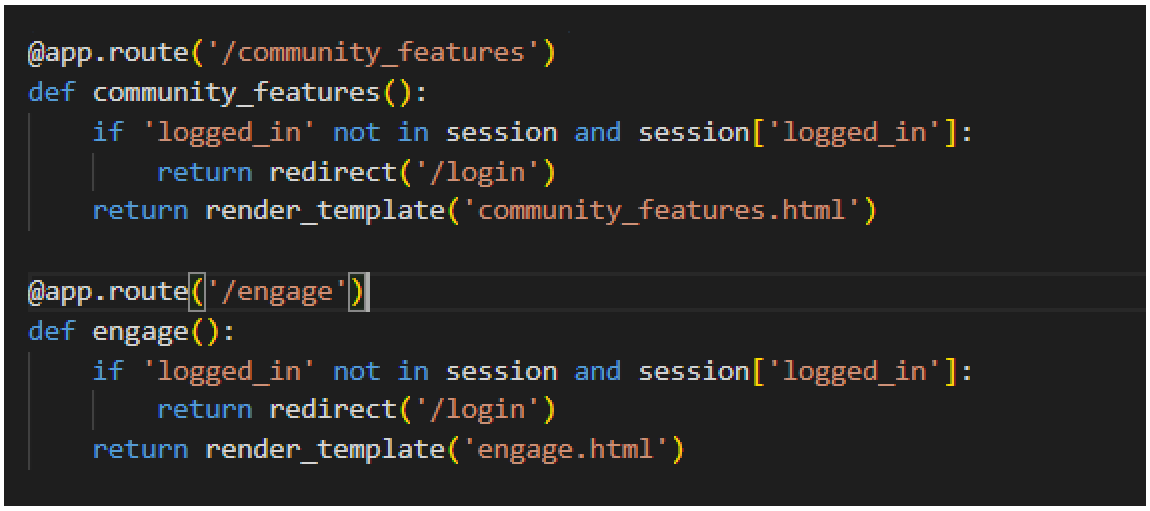
3. Community Features Route (/community_features)

- **Purpose**: Renders a page for community features of the application.
- **Functionality**:
  - Checks if the user is logged in. If not, it redirects to the login page.
  - If logged in, it renders the community_features.html template.

Summary

This section of the code focuses on automating workout reminders via email, ensuring users are prompted to engage with their fitness routines. The functionality is backed by scheduled jobs that fetch user data from the database, making the process seamless and user-friendly. Additionally, the community features route enhances user engagement by providing access to relevant features in the application.

Overview of Community Features and Engagement Functionality



```python
@app.route('/community_features')
def community_features():
    if 'logged_in' not in session and session['logged_in']:
        return redirect('/login')
    return render_template('community_features.html')

@app.route('/engage')
def engage():
    if 'logged_in' not in session and session['logged_in']:
        return redirect('/login')
    return render_template('engage.html')
```

1. Community Features Route (/community_features)

- **Purpose**: This route is designed to display the community features of the application to logged-in users.

- **Functionality**:
    - **Login Check**: Verifies if the user is logged in by checking for the 'logged_in' key in the session. Note that the condition should be if 'logged_in' not in session or not session['logged_in']: to ensure it properly checks the login status.
    - **Redirect on Not Logged In**: If the user is not logged in, they are redirected to the login page.
    - **Render Template**: If the user is logged in, it renders the community_features.html template, which presumably contains content related to community engagement within the gym organization.

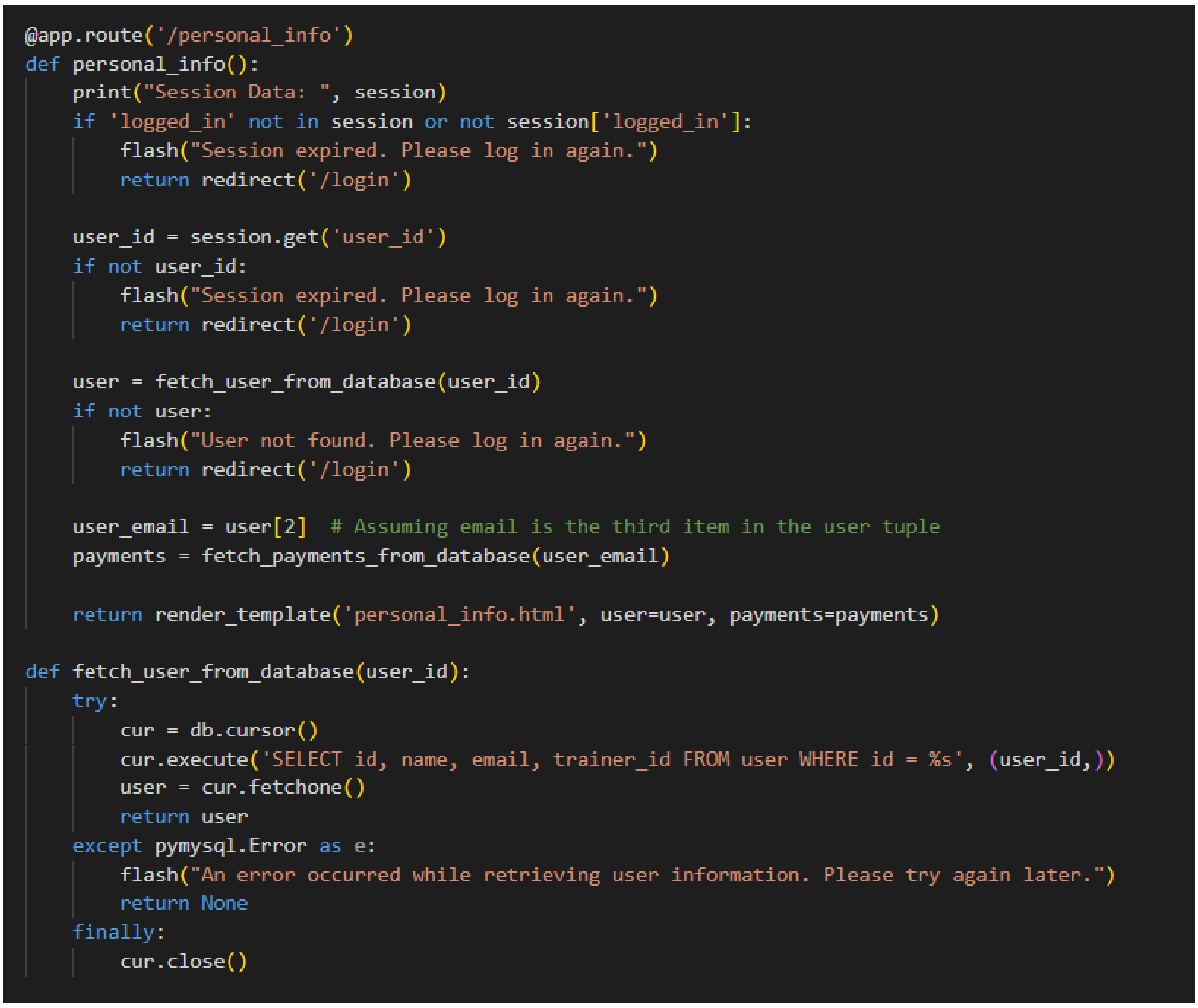
2. Engagement Route (/engage)

- **Purpose**: This route allows users to access engagement-related features or content.
- **Functionality**:
    - **Login Check**: Similar to the community features route, it checks if the user is logged in using the session variable.
    - **Redirect on Not Logged In**: Redirects to the login page if the user is not authenticated.
    - **Render Template**: If logged in, it renders the engage.html template, which likely contains interactive elements or features aimed at increasing user involvement and engagement within the community.

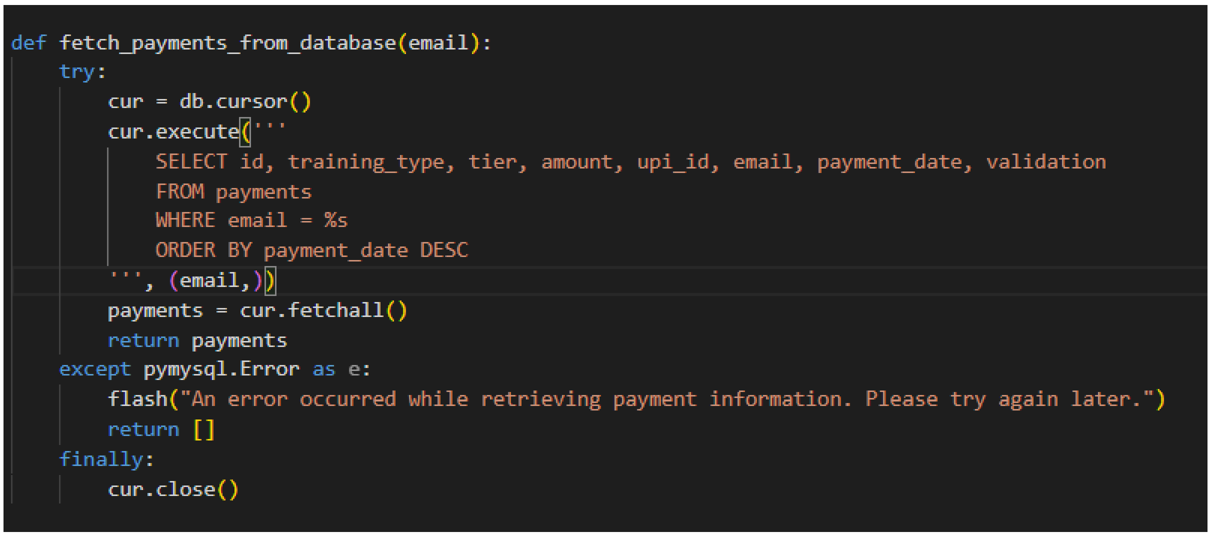
Summary

These routes serve as essential components for user engagement and community interaction in the application. By ensuring users are authenticated before accessing community features and engagement tools, the application promotes a secure environment while encouraging active participation among gym members. The use of template rendering allows for dynamic content delivery based on user interactions.

Overview of Personal Information and Payment Retrieval Functionality



```python
@app.route('/personal_info')
def personal_info():
    print("Session Data: ", session)
    if 'logged_in' not in session or not session['logged_in']:
        flash("Session expired. Please log in again.")
        return redirect('/login')

    user_id = session.get('user_id')
    if not user_id:
        flash("Session expired. Please log in again.")
        return redirect('/login')

    user = fetch_user_from_database(user_id)
    if not user:
        flash("User not found. Please log in again.")
        return redirect('/login')

    user_email = user[2]  # Assuming email is the third item in the user tuple
    payments = fetch_payments_from_database(user_email)

    return render_template('personal_info.html', user=user, payments=payments)

def fetch_user_from_database(user_id):
    try:
        cur = db.cursor()
        cur.execute('SELECT id, name, email, trainer_id FROM user WHERE id = %s', (user_id,))
        user = cur.fetchone()
        return user
    except pymysql.Error as e:
        flash("An error occurred while retrieving user information. Please try again later.")
        return None
    finally:
        cur.close()
```



```python
def fetch_payments_from_database(email):
    try:
        cur = db.cursor()
        cur.execute('''
            SELECT id, training_type, tier, amount, upi_id, email, payment_date, validation
            FROM payments
            WHERE email = %s
            ORDER BY payment_date DESC
        ''', (email,))
        payments = cur.fetchall()
        return payments
    except pymysql.Error as e:
        flash("An error occurred while retrieving payment information. Please try again later.")
        return []
    finally:
        cur.close()
```

1. Personal Information Route (/personal_info)

- **Purpose**: This route retrieves and displays the logged-in user's personal information and payment history.

- **Functionality**:
    - **Session Data Logging**: The current session data is printed for debugging purposes.
    - **Login Check**:
        - It checks if the user is logged in by verifying the presence of the 'logged_in' key in the session and its truthiness.
        - If not logged in, a flash message is displayed indicating the session has expired, and the user is redirected to the login page.
    - **User ID Check**:
        - It retrieves the user_id from the session. If not found, it prompts the user to log in again.
    - **User Fetching**:
        - Calls fetch_user_from_database(user_id) to retrieve the user's details.
        - If no user is found, it displays a flash message and redirects to the login page.
    - **Payment Retrieval**:
        - After successfully fetching the user, it retrieves their email and calls fetch_payments_from_database(user_email) to get their payment history.
    - **Render Template**: The personal_info.html template is rendered with the user and payments data.

2. Fetching User from Database (fetch_user_from_database)

- **Purpose**: To retrieve the user's details based on their user_id.
- **Parameters**:
    - user_id: The ID of the user whose information needs to be fetched.
- **Functionality**:
    - **Database Interaction**: Executes a SQL query to select the user's ID, name, email, and trainer ID.
    - **Error Handling**: Catches any database-related errors and displays a flash message.
    - **Return Value**: Returns the user tuple if found, otherwise returns None.

3. Fetching Payments from Database (fetch_payments_from_database)

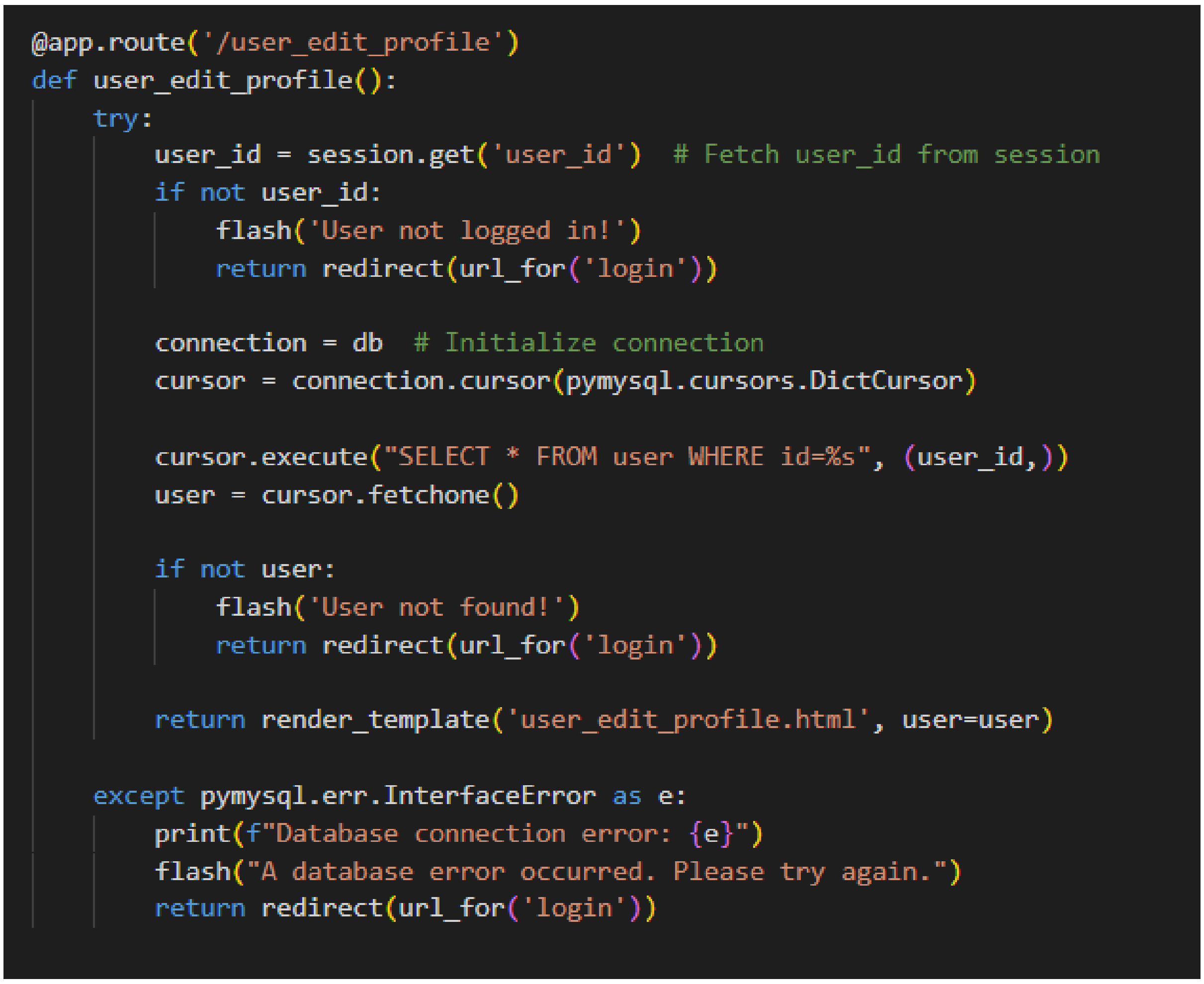
- **Purpose**: To retrieve the payment history for a specific user based on their email.
- **Parameters**:

    - email: The email of the user whose payment records need to be retrieved.
- **Functionality**:
    - **Database Interaction**: Executes a SQL query to select payment records, ordered by payment date in descending order.
    - **Error Handling**: Catches any database-related errors and displays a flash message. Returns an empty list if an error occurs.
    - **Return Value**: Returns a list of payment records.

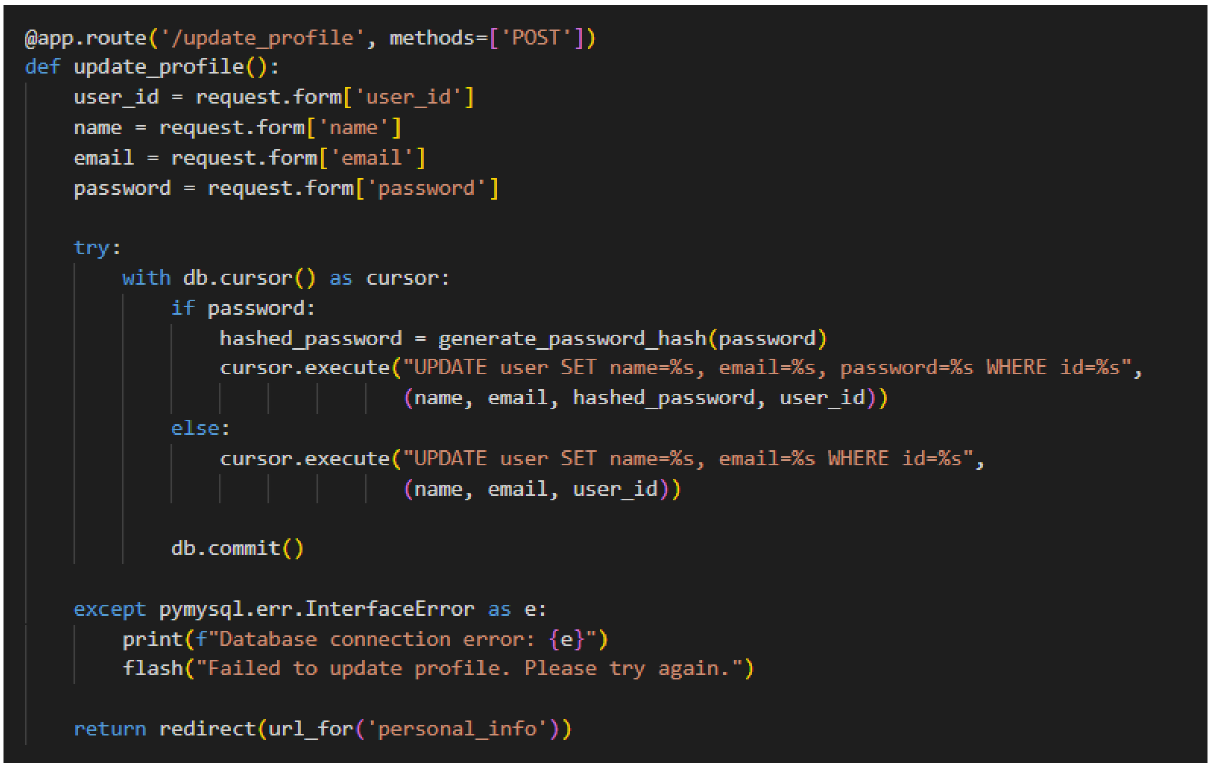
Summary

This functionality ensures that users can access their personal information and payment history securely. By implementing robust session checks and database retrieval methods, the application maintains user security and provides valuable insights into user data. The separation of concerns in data fetching through dedicated functions promotes better organization and readability of the code.

Overview of User Profile Editing Functionality



```python
@app.route('/user_edit_profile')
def user_edit_profile():
    try:
        user_id = session.get('user_id')  # Fetch user_id from session
        if not user_id:
            flash('User not logged in!')
            return redirect(url_for('login'))

        connection = db  # Initialize connection
        cursor = connection.cursor(pymysql.cursors.DictCursor)

        cursor.execute("SELECT * FROM user WHERE id=%s", (user_id,))
        user = cursor.fetchone()

        if not user:
            flash('User not found!')
            return redirect(url_for('login'))

        return render_template('user_edit_profile.html', user=user)

    except pymysql.err.InterfaceError as e:
        print(f"Database connection error: {e}")
        flash("A database error occurred. Please try again.")
        return redirect(url_for('login'))
```



```python
@app.route('/update_profile', methods=['POST'])
def update_profile():
    user_id = request.form['user_id']
    name = request.form['name']
    email = request.form['email']
    password = request.form['password']

    try:
        with db.cursor() as cursor:
            if password:
                hashed_password = generate_password_hash(password)
                cursor.execute("UPDATE user SET name=%s, email=%s, password=%s WHERE id=%s",
                               (name, email, hashed_password, user_id))
            else:
                cursor.execute("UPDATE user SET name=%s, email=%s WHERE id=%s",
                               (name, email, user_id))

            db.commit()

    except pymysql.err.InterfaceError as e:
        print(f"Database connection error: {e}")
        flash("Failed to update profile. Please try again.")

    return redirect(url_for('personal_info'))
```

1. Edit User Profile Route (/user_edit_profile)

- **Purpose**: This route allows a logged-in user to access their profile for editing.
- **Functionality**:
    - **Session Validation**:
        - The user ID is retrieved from the session to ensure the user is logged in.
        - If the user ID is not found, a flash message is displayed, and the user is redirected to the login page.
    - **Database Interaction**:
        - Establishes a connection to the database and creates a cursor.
        - Executes a SQL query to fetch the user's details based on their ID.
        - If no user is found, a flash message is shown, and the user is redirected to the login page.
    - **Render Template**: If the user is found, it renders the user_edit_profile.html template, passing the user data to the template.

2. Update User Profile Route (/update_profile)

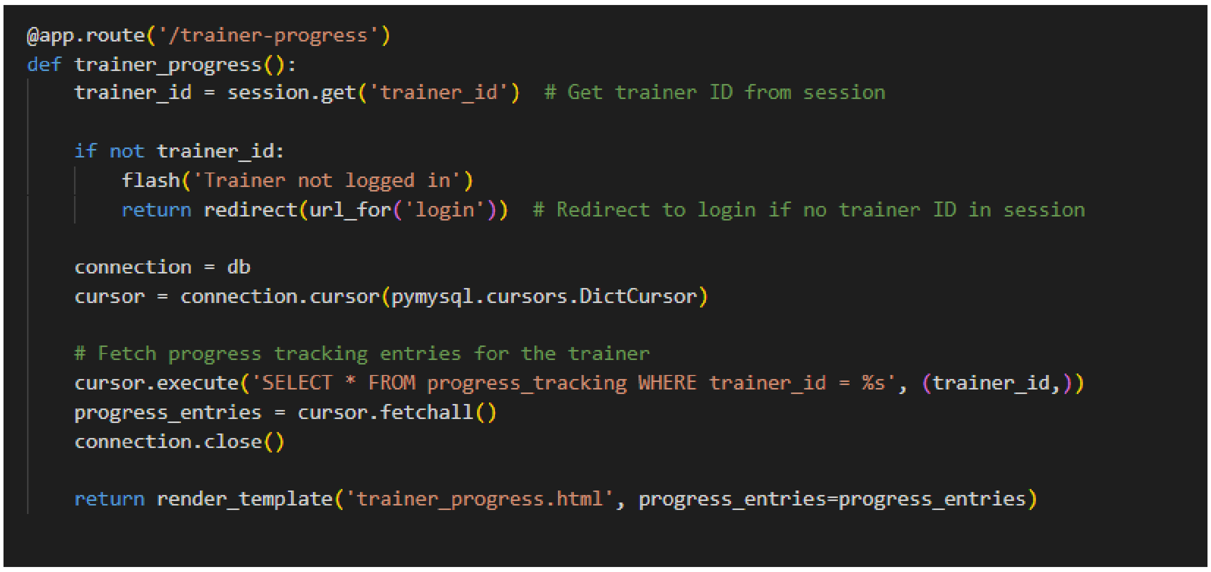
- **Purpose**: This route handles the profile update request from the user.
- **Parameters**: The function captures the user's ID, name, email, and password from the submitted form data.

- **Functionality**:
    - **Database Update**:
        - A database cursor is created to perform the update operation.
        - If a new password is provided, it hashes the password using generate_password_hash and updates the user details in the database, including the new password.
        - If no password is provided, it updates only the name and email.
    - **Error Handling**:
        - Catches any InterfaceError from the database connection and displays a flash message indicating the failure of the profile update.
    - **Redirection**: After attempting to update, the user is redirected back to their personal information page.

Summary

This functionality ensures that users can edit and update their profile information securely. By validating user sessions and handling database interactions robustly, the application maintains user security and data integrity. The separation of logic into dedicated routes allows for better organization and maintainability of the code.

Overview of Trainer Progress Functionality



```
@app.route('/trainer-progress')
def trainer_progress():
    trainer_id = session.get('trainer_id')  # Get trainer ID from session

    if not trainer_id:
        flash('Trainer not logged in')
        return redirect(url_for('login'))  # Redirect to login if no trainer ID in session

    connection = db
    cursor = connection.cursor(pymysql.cursors.DictCursor)

    # Fetch progress tracking entries for the trainer
    cursor.execute('SELECT * FROM progress_tracking WHERE trainer_id = %s', (trainer_id,))
    progress_entries = cursor.fetchall()
    connection.close()

    return render_template('trainer_progress.html', progress_entries=progress_entries)
```

1. Trainer Progress Route (/trainer-progress)

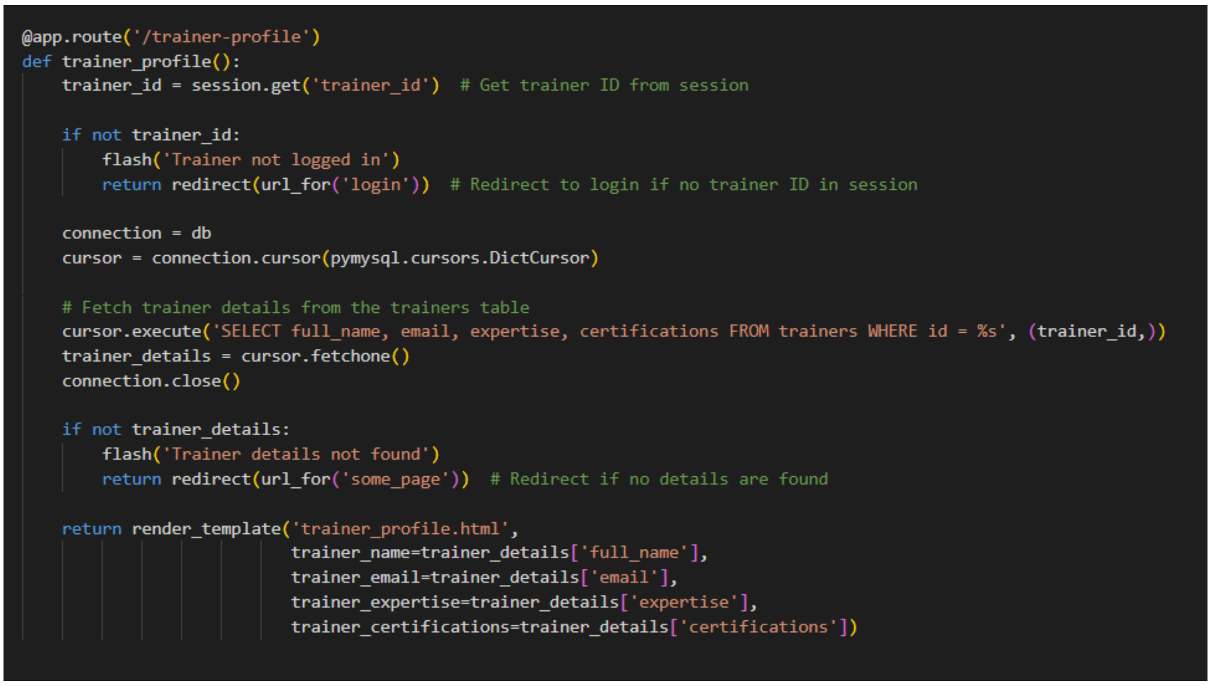
- **Purpose**: This route allows logged-in trainers to view their progress tracking entries.
- **Functionality**:
    - **Session Validation**:

    - Retrieves the trainer ID from the session to ensure the trainer is logged in.
    - If the trainer ID is not present, it triggers a flash message indicating that the trainer is not logged in and redirects them to the login page.
  - **Database Interaction**:
    - Establishes a connection to the database and creates a cursor with a dictionary cursor type for easier access to column names.
    - Executes a SQL query to fetch all progress tracking entries associated with the logged-in trainer's ID.
    - Closes the database connection after fetching the data to free up resources.
  - **Render Template**:
    - If progress entries are retrieved successfully, it renders the trainer_progress.html template and passes the fetched progress entries to it for display.

Summary

This functionality ensures that trainers can access and view their progress tracking information securely. By validating the trainer's login status and efficiently querying the database, it provides a seamless user experience. The organized structure of the code supports maintainability and clarity, making it easier to manage and extend in the future.

Overview of Trainer Profile Functionality



```python
@app.route('/trainer-profile')
def trainer_profile():
    trainer_id = session.get('trainer_id')  # Get trainer ID from session

    if not trainer_id:
        flash('Trainer not logged in')
        return redirect(url_for('login'))  # Redirect to login if no trainer ID in session

    connection = db
    cursor = connection.cursor(pymysql.cursors.DictCursor)

    # Fetch trainer details from the trainers table
    cursor.execute('SELECT full_name, email, expertise, certifications FROM trainers WHERE id = %s', (trainer_id,))
    trainer_details = cursor.fetchone()
    connection.close()

    if not trainer_details:
        flash('Trainer details not found')
        return redirect(url_for('some_page'))  # Redirect if no details are found

    return render_template('trainer_profile.html',
                           trainer_name=trainer_details['full_name'],
                           trainer_email=trainer_details['email'],
                           trainer_expertise=trainer_details['expertise'],
                           trainer_certifications=trainer_details['certifications'])
```

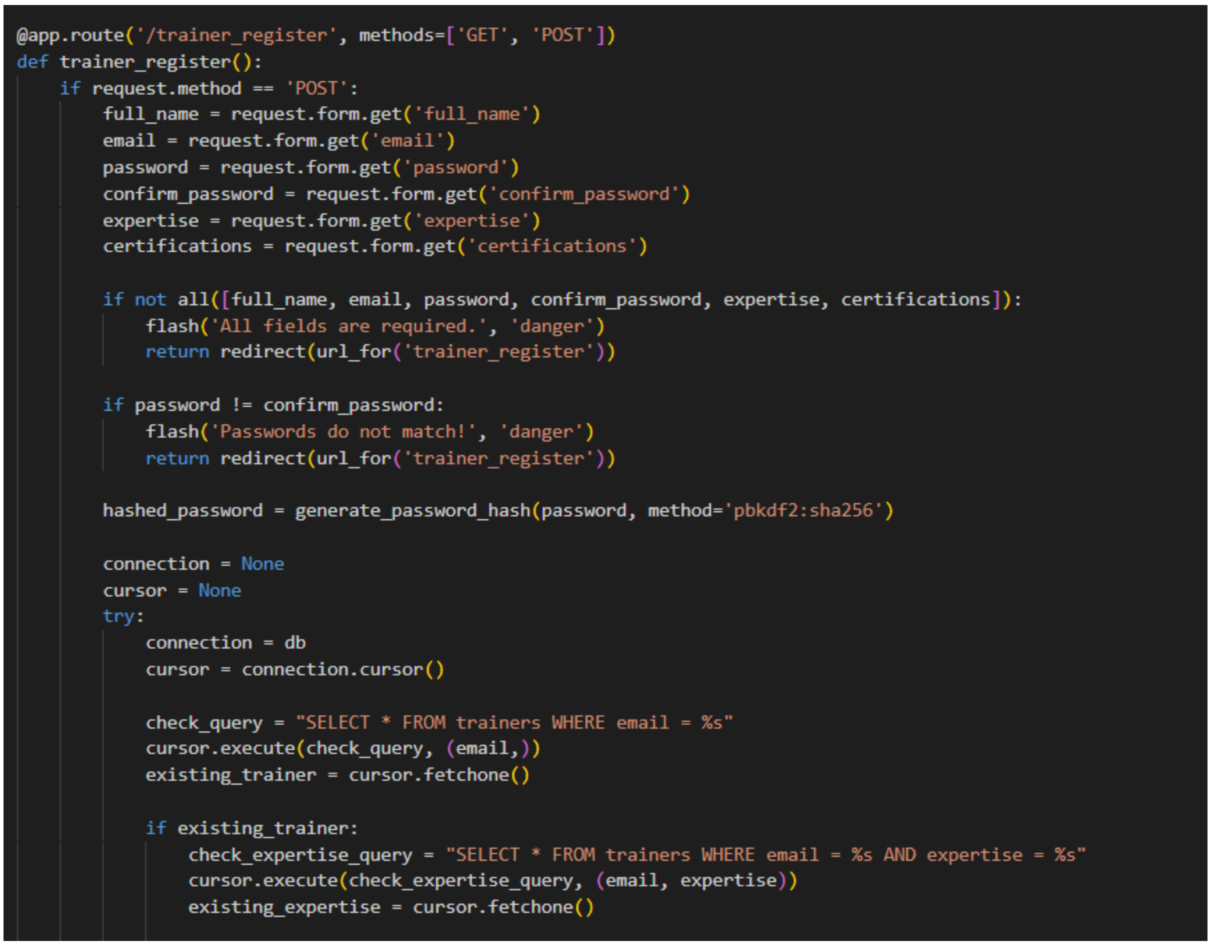
1. Trainer Profile Route (/trainer-profile)

- **Purpose**: This route allows trainers to view their profile details.
- **Functionality**:
    - **Session Validation**:
        - Retrieves the trainer ID from the session to confirm that the trainer is logged in.
        - If the trainer ID is absent, it displays a flash message indicating that the trainer is not logged in and redirects them to the login page.
    - **Database Interaction**:
        - Establishes a connection to the database and creates a cursor using a dictionary cursor type for easy access to column names.
        - Executes a SQL query to fetch the trainer's full name, email, expertise, and certifications from the trainers table based on the logged-in trainer's ID.
        - Closes the database connection after the query to release resources.
    - **Error Handling**:
        - If no trainer details are found, it flashes a message stating "Trainer details not found" and redirects the user to a specified page.
    - **Render Template**:
        - If trainer details are successfully retrieved, it renders the trainer_profile.html template, passing the trainer's information to be displayed.

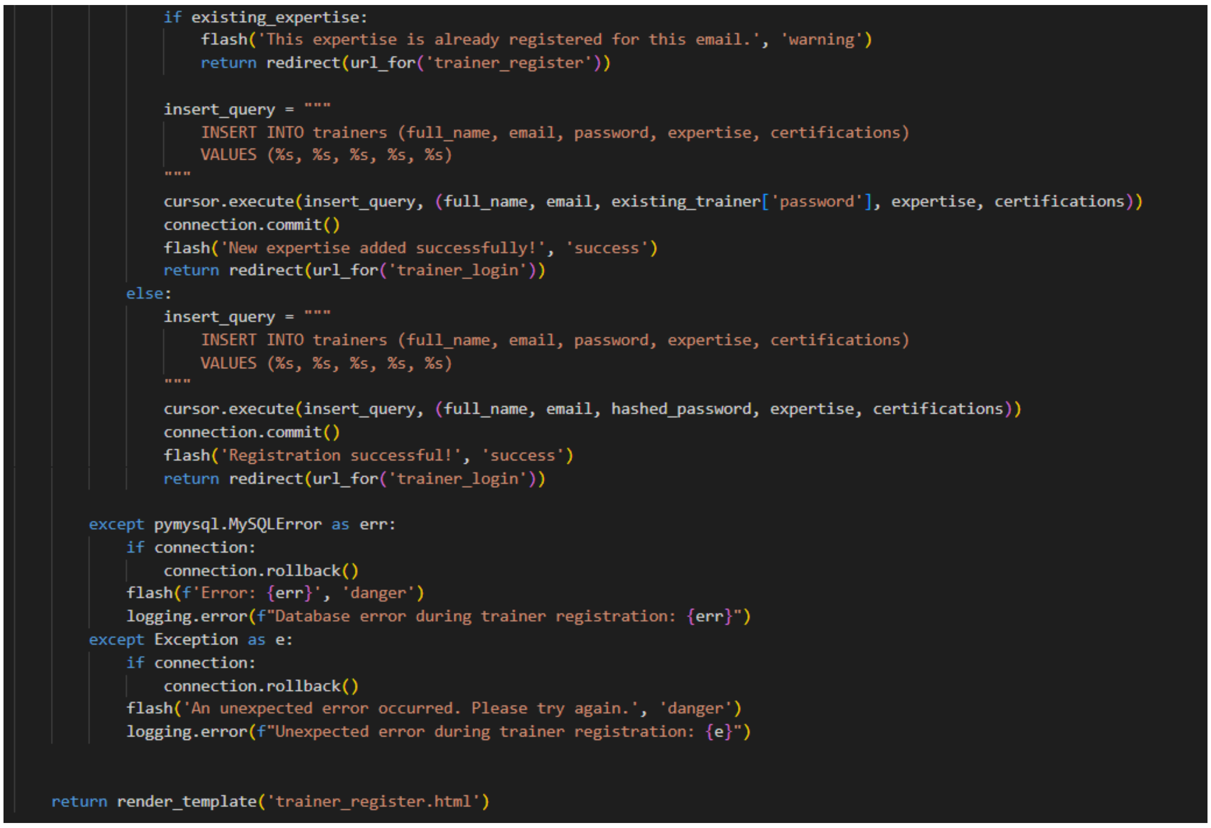
Summary

This functionality provides trainers with access to their personal profile information in a secure manner. By ensuring the trainer is logged in and querying the database efficiently, it enhances user experience. The structured approach in error handling and data retrieval maintains code readability and allows for future scalability.

Overview of Trainer Registration Functionality



```python
@app.route('/trainer_register', methods=['GET', 'POST'])
def trainer_register():
    if request.method == 'POST':
        full_name = request.form.get('full_name')
        email = request.form.get('email')
        password = request.form.get('password')
        confirm_password = request.form.get('confirm_password')
        expertise = request.form.get('expertise')
        certifications = request.form.get('certifications')

        if not all([full_name, email, password, confirm_password, expertise, certifications]):
            flash('All fields are required.', 'danger')
            return redirect(url_for('trainer_register'))

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

        hashed_password = generate_password_hash(password, method='pbkdf2:sha256')

        connection = None
        cursor = None
        try:
            connection = db
            cursor = connection.cursor()

            check_query = "SELECT * FROM trainers WHERE email = %s"
            cursor.execute(check_query, (email,))
            existing_trainer = cursor.fetchone()

            if existing_trainer:
                check_expertise_query = "SELECT * FROM trainers WHERE email = %s AND expertise = %s"
                cursor.execute(check_expertise_query, (email, expertise))
                existing_expertise = cursor.fetchone()
```



```python
                if existing_expertise:
                    flash('This expertise is already registered for this email.', 'warning')
                    return redirect(url_for('trainer_register'))

                insert_query = """
                    INSERT INTO trainers (full_name, email, password, expertise, certifications)
                    VALUES (%s, %s, %s, %s, %s)
                """
                cursor.execute(insert_query, (full_name, email, existing_trainer['password'], expertise, certifications))
                connection.commit()
                flash('New expertise added successfully!', 'success')
                return redirect(url_for('trainer_login'))
            else:
                insert_query = """
                    INSERT INTO trainers (full_name, email, password, expertise, certifications)
                    VALUES (%s, %s, %s, %s, %s)
                """
                cursor.execute(insert_query, (full_name, email, hashed_password, expertise, certifications))
                connection.commit()
                flash('Registration successful!', 'success')
                return redirect(url_for('trainer_login'))

        except pymysql.MySQLError as err:
            if connection:
                connection.rollback()
            flash(f'Error: {err}', 'danger')
            logging.error(f"Database error during trainer registration: {err}")
        except Exception as e:
            if connection:
                connection.rollback()
            flash('An unexpected error occurred. Please try again.', 'danger')
            logging.error(f"Unexpected error during trainer registration: {e}")


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

1. Trainer Registration Route (/trainer_register)

- **Purpose**: This route handles the registration process for trainers, including both displaying the registration form and processing the registration data upon submission.
- **Functionality**:

Form Handling:

- **GET Method**: Renders the registration form when accessed with a GET request.
- **POST Method**: Processes the registration data when the form is submitted.

Form Validation:

- Checks if all required fields (full name, email, password, confirm password, expertise, certifications) are filled. If any field is empty, a flash message is displayed, and the user is redirected back to the registration form.
- Validates that the password and confirm password match. If they do not, a flash message is shown, and the user is redirected back to the registration form.

Password Hashing:

- If the password is valid, it is hashed using generate_password_hash for security.

Database Interaction:

    - Establishes a connection to the database.
    - **Check for Existing Trainer**: Executes a query to check if a trainer with the same email already exists.
        - If the email exists, another query checks if the same expertise is registered for that email.
            - If it is, a flash message is displayed, indicating that the expertise is already registered, and the user is redirected back.
        - If the email exists but with different expertise, it inserts a new entry for the new expertise without changing the existing password.
    - If the email does not exist, it inserts a new trainer record with the provided details.

Error Handling:

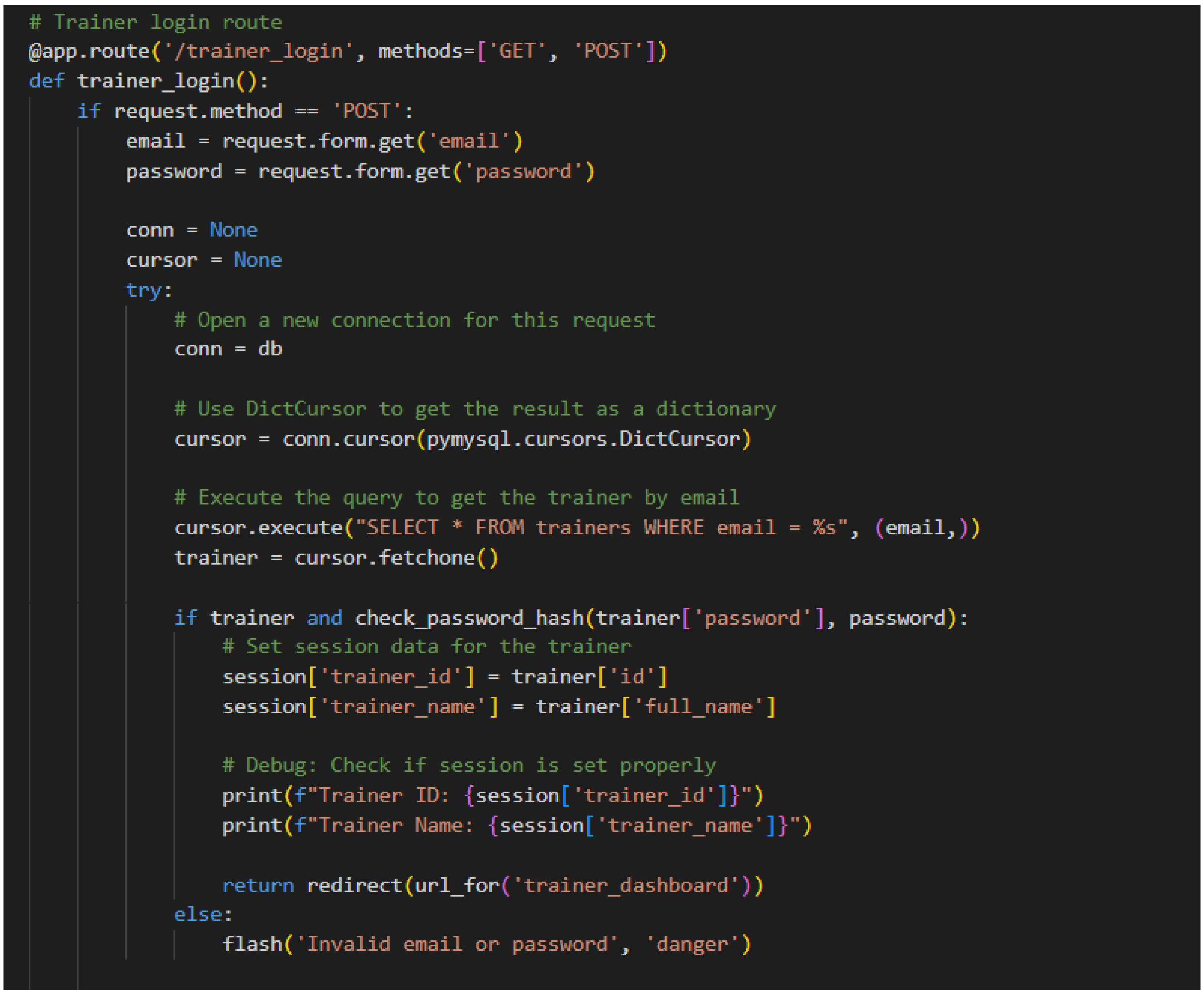
    - Any database errors encountered during the process trigger a rollback to ensure data integrity.
    - Flash messages are shown to the user for errors, and detailed error logs are generated for troubleshooting.

- **Render Template**: If the request method is GET, it renders the trainer_register.html template for the registration form.

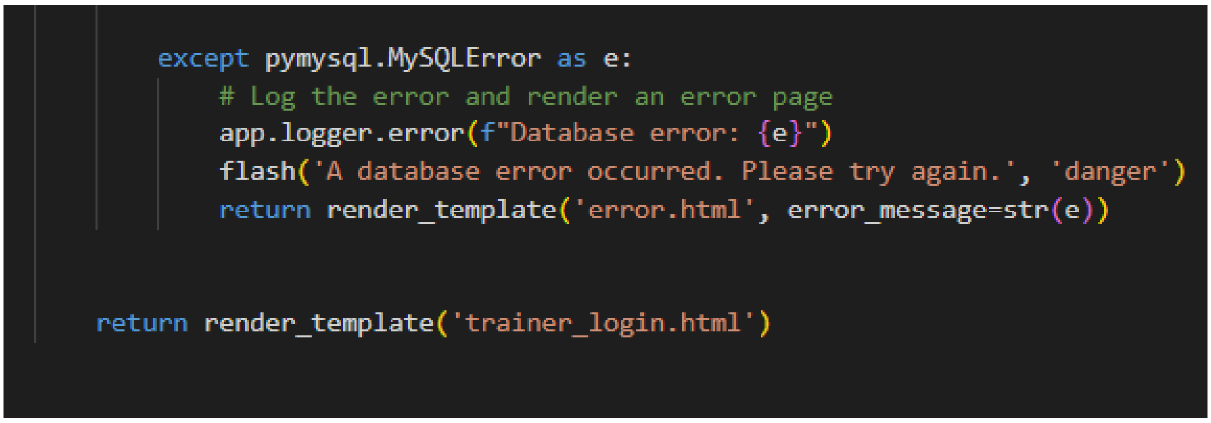
Summary

This functionality provides a secure and user-friendly registration process for trainers. It ensures that all required information is validated, handles duplicate registrations, and securely stores trainer credentials. Proper error handling and logging enhance the robustness of the registration process, making it easier to maintain and troubleshoot.

Overview of Trainer Login Functionality



```python
# Trainer login route
@app.route('/trainer_login', methods=['GET', 'POST'])
def trainer_login():
    if request.method == 'POST':
        email = request.form.get('email')
        password = request.form.get('password')

        conn = None
        cursor = None
        try:
            # Open a new connection for this request
            conn = db

            # Use DictCursor to get the result as a dictionary
            cursor = conn.cursor(pymysql.cursors.DictCursor)

            # Execute the query to get the trainer by email
            cursor.execute("SELECT * FROM trainers WHERE email = %s", (email,))
            trainer = cursor.fetchone()

            if trainer and check_password_hash(trainer['password'], password):
                # Set session data for the trainer
                session['trainer_id'] = trainer['id']
                session['trainer_name'] = trainer['full_name']

                # Debug: Check if session is set properly
                print(f"Trainer ID: {session['trainer_id']}")
                print(f"Trainer Name: {session['trainer_name']}")

                return redirect(url_for('trainer_dashboard'))
            else:
                flash('Invalid email or password', 'danger')
```



```python
        except pymysql.MySQLError as e:
            # Log the error and render an error page
            app.logger.error(f"Database error: {e}")
            flash('A database error occurred. Please try again.', 'danger')
            return render_template('error.html', error_message=str(e))


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

1. Trainer Login Route (/trainer_login)

- **Purpose**: This route manages the login process for trainers, allowing them to access their dashboard upon successful authentication.
- **Functionality**:

Form Handling:

  - **GET Method**: Renders the login form when accessed via a GET request.

    - **POST Method**: Processes the login credentials when the form is submitted.

Login Credential Validation:

    - Retrieves the trainer's email and password from the form.
    - Establishes a database connection to validate the credentials.
    - Uses DictCursor to fetch results as dictionaries for easy access to column names.

Database Interaction:

    - Executes a query to fetch the trainer record based on the provided email.
    - Checks if a trainer with the given email exists and if the provided password matches the hashed password stored in the database using check_password_hash.

Session Management:

    - If the credentials are valid, session data is set for the trainer (trainer ID and name), allowing them to be recognized in subsequent requests.
    - Debugging print statements confirm that session data is set properly.

Error Handling:

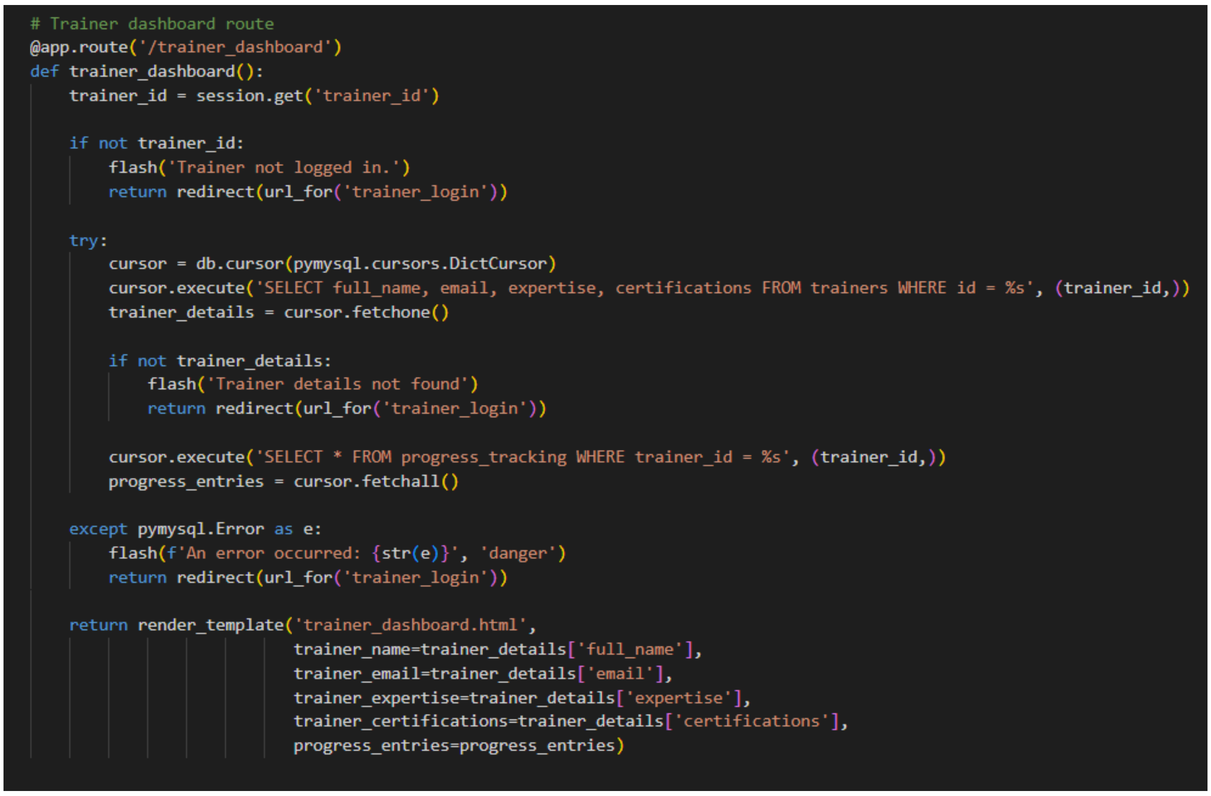
    - If the email or password is incorrect, a flash message is displayed to the user indicating that the login credentials are invalid.
    - Any database errors encountered trigger logging of the error and rendering of an error page, along with a flash message to inform the user of the issue.

- **Render Template**: If the request method is GET, the login form is displayed via the trainer_login.html template.

Summary

This login functionality offers a straightforward and secure way for trainers to access their dashboard. It emphasizes credential validation, session management, and error handling, ensuring that trainers can log in successfully while providing helpful feedback in case of errors. The use of hashed passwords enhances security, while structured error handling aids in maintaining the integrity of the application.

Overview of Trainer Dashboard Functionality



```python
# Trainer dashboard route
@app.route('/trainer_dashboard')
def trainer_dashboard():
    trainer_id = session.get('trainer_id')

    if not trainer_id:
        flash('Trainer not logged in.')
        return redirect(url_for('trainer_login'))

    try:
        cursor = db.cursor(pymysql.cursors.DictCursor)
        cursor.execute('SELECT full_name, email, expertise, certifications FROM trainers WHERE id = %s', (trainer_id,))
        trainer_details = cursor.fetchone()

        if not trainer_details:
            flash('Trainer details not found')
            return redirect(url_for('trainer_login'))

        cursor.execute('SELECT * FROM progress_tracking WHERE trainer_id = %s', (trainer_id,))
        progress_entries = cursor.fetchall()

    except pymysql.Error as e:
        flash(f'An error occurred: {str(e)}', 'danger')
        return redirect(url_for('trainer_login'))

    return render_template('trainer_dashboard.html',
                           trainer_name=trainer_details['full_name'],
                           trainer_email=trainer_details['email'],
                           trainer_expertise=trainer_details['expertise'],
                           trainer_certifications=trainer_details['certifications'],
                           progress_entries=progress_entries)
```

1. Trainer Dashboard Route (/trainer_dashboard)

- **Purpose**: This route serves as the main dashboard for trainers, displaying their profile details and progress tracking entries.
- **Functionality**:

Session Validation:

  - The route checks if the trainer_id exists in the session, which indicates that the trainer is logged in. If not, a flash message is displayed, and the trainer is redirected to the login page.

Database Interaction:

  - Establishes a connection to the database and retrieves the trainer's details (full name, email, expertise, certifications) based on the trainer_id.
  - Executes a second query to fetch all progress tracking entries associated with the trainer.

Error Handling:

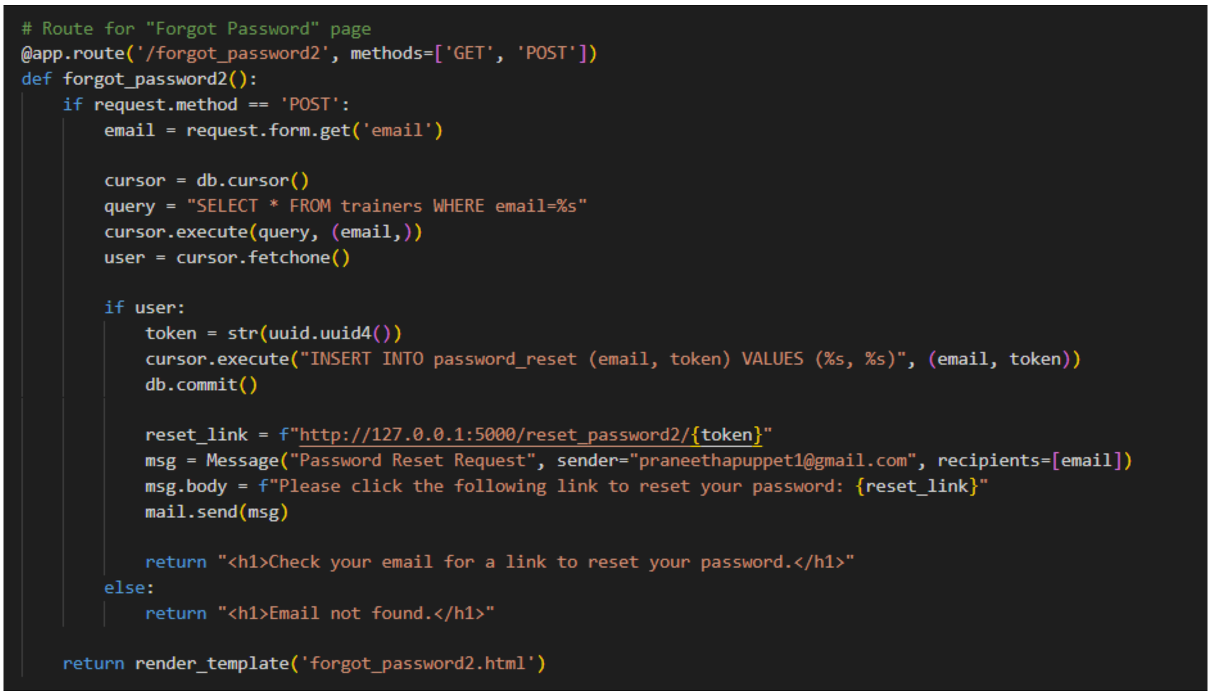
  - If there is an issue retrieving trainer details or progress entries, an error message is displayed, and the trainer is redirected back to the login page.
  - Uses pymysql.Error to catch any database-related issues.

- **Render Template**:
    - If all queries are successful, the trainer_dashboard.html template is rendered with the trainer's details and their progress tracking entries passed as context variables.

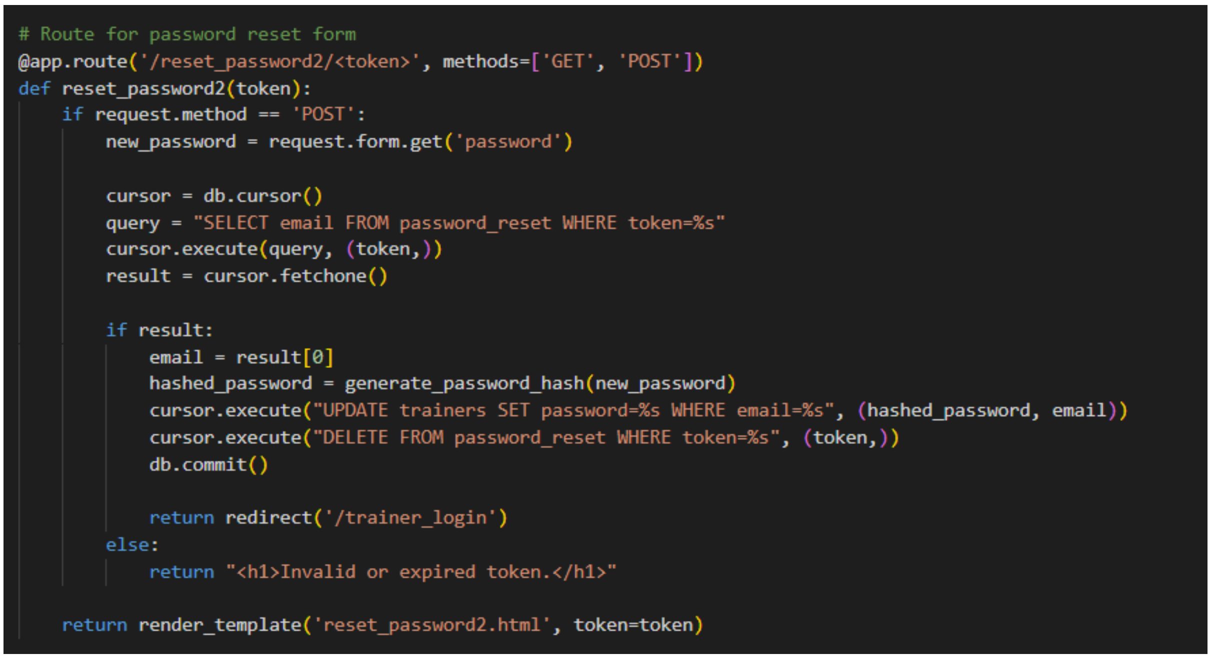
Summary

This dashboard route provides trainers with a comprehensive overview of their profile and any relevant progress entries. It ensures secure access by validating the session, retrieves essential information from the database, and handles potential errors gracefully. The user-friendly feedback through flash messages enhances the user experience, guiding trainers through any issues that may arise.

Overview of Forgot Password Functionality



```python
# Route for "Forgot Password" page
@app.route('/forgot_password2', methods=['GET', 'POST'])
def forgot_password2():
    if request.method == 'POST':
        email = request.form.get('email')

        cursor = db.cursor()
        query = "SELECT * FROM trainers WHERE email=%s"
        cursor.execute(query, (email,))
        user = cursor.fetchone()

        if user:
            token = str(uuid.uuid4())
            cursor.execute("INSERT INTO password_reset (email, token) VALUES (%s, %s)", (email, token))
            db.commit()

            reset_link = f"http://127.0.0.1:5000/reset_password2/{token}"
            msg = Message("Password Reset Request", sender="praneethapuppet1@gmail.com", recipients=[email])
            msg.body = f"Please click the following link to reset your password: {reset_link}"
            mail.send(msg)

            return "<h1>Check your email for a link to reset your password.</h1>"
        else:
            return "<h1>Email not found.</h1>"

    return render_template('forgot_password2.html')
```



```
# Route for password reset form
@app.route('/reset_password2/<token>', methods=['GET', 'POST'])
def reset_password2(token):
    if request.method == 'POST':
        new_password = request.form.get('password')

        cursor = db.cursor()
        query = "SELECT email FROM password_reset WHERE token=%s"
        cursor.execute(query, (token,))
        result = cursor.fetchone()

        if result:
            email = result[0]
            hashed_password = generate_password_hash(new_password)
            cursor.execute("UPDATE trainers SET password=%s WHERE email=%s", (hashed_password, email))
            cursor.execute("DELETE FROM password_reset WHERE token=%s", (token,))
            db.commit()

            return redirect('/trainer_login')
        else:
            return "<h1>Invalid or expired token.</h1>"

    return render_template('reset_password2.html', token=token)
```

1. Forgot Password Route (/forgot_password2)

- **Purpose**: This route handles the process of initiating a password reset request for trainers who have forgotten their passwords.
- **Functionality**:

POST Request Handling:

  - When the form is submitted, the route retrieves the email provided by the user.
  - It queries the database to check if the email exists in the trainers table.

Token Generation and Email Sending:

  - If the email is found, a unique token is generated using uuid.
  - The token and email are inserted into a password_reset table to track the reset request.
  - A reset link is constructed, which includes the token, and an email is sent to the trainer with this link using Flask-Mail.

Feedback to User:

  - A success message is returned if the email is found and the reset link is sent; otherwise, a message indicates that the email was not found.

- **Render Template**:
  - If the request method is GET, the forgot_password2.html template is rendered for the user to enter their email.

2. Reset Password Route (/reset_password2/<token>)

- **Purpose**: This route allows trainers to reset their password using the token sent to their email.
- **Functionality**:

POST Request Handling:

- When the form is submitted, the new password is retrieved from the request.
- The route checks if the provided token is valid by querying the password_reset table.

Password Update:

- If the token is valid, the corresponding email is fetched, and the trainer's password is updated in the trainers table.
- The reset token is then deleted from the password_reset table to prevent reuse.

Redirect:

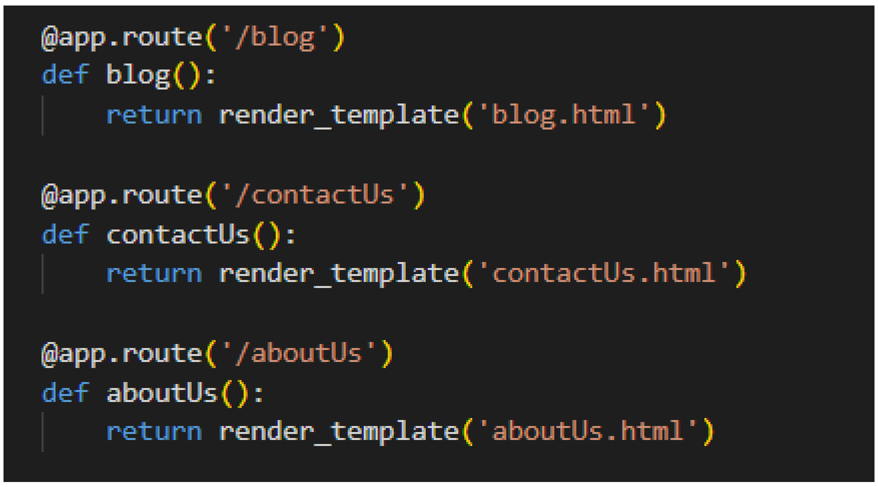
- After successfully resetting the password, the user is redirected to the login page.

- **Render Template**:
    - If the request method is GET, the reset_password2.html template is rendered, allowing the user to enter a new password.

Summary

This implementation allows trainers to securely reset their passwords by sending a unique link via email. The use of tokens helps ensure that the reset process is secure, while database interactions manage the necessary state transitions effectively. The feedback provided to users guides them through the process, enhancing the overall user experience.

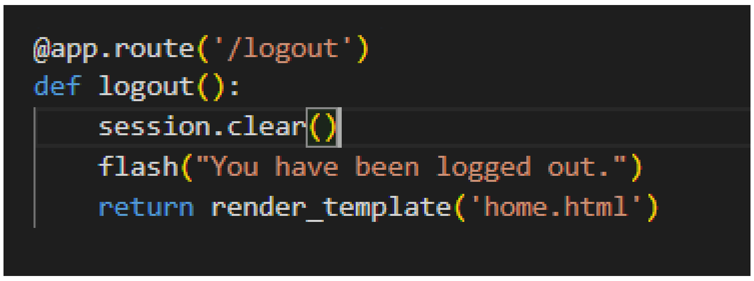
Routes Overview



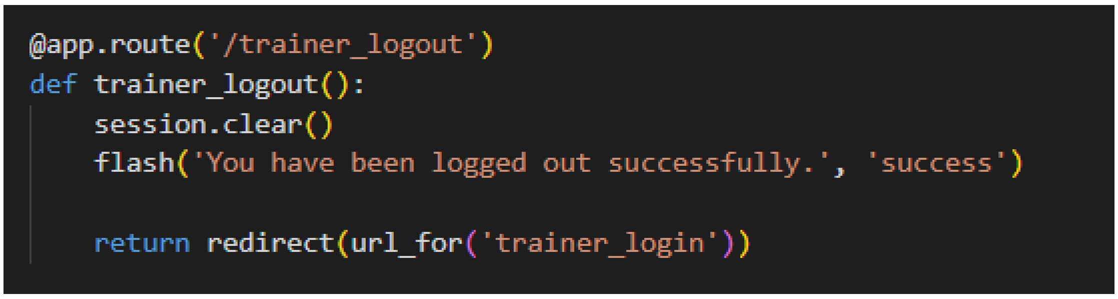
```python
@app.route('/blog')
def blog():
    return render_template('blog.html')

@app.route('/contactUs')
def contactUs():
    return render_template('contactUs.html')

@app.route('/aboutUs')
def aboutUs():
    return render_template('aboutUs.html')
```

1. **Blog Route (/blog)**
    - **Functionality**: Renders the blog.html template.
    - **Usage**: This is where users can view blog posts or articles.
2. **Contact Us Route (/contactUs)**
    - **Functionality**: Renders the contactUs.html template.
    - **Usage**: This page can be used for users to submit inquiries or get in touch with your organization.
3. **About Us Route (/aboutUs)**
    - **Functionality**: Renders the aboutUs.html template.
    - **Usage**: This page typically contains information about your organization, its mission, and its team.

Logout Functionality



```
@app.route('/logout')
def logout():
    session.clear()
    flash("You have been logged out.")
    return render_template('home.html')
```



```
@app.route('/trainer_logout')
def trainer_logout():
    session.clear()
    flash('You have been logged out successfully.', 'success')

    return redirect(url_for('trainer_login'))
```

1. Trainer Logout Route (/trainer_logout)

- **Purpose**: This route is designed to log out a trainer from the application.
- **Functionality**:
    - It clears all session data using session.clear(), effectively logging the trainer out.
    - Displays a success message indicating that the trainer has been logged out successfully.

2. General Logout Route (/logout)

- **Purpose**: This route serves as a general logout option for all users (e.g., trainers, members).
- **Functionality**:
    - Similar to the trainer logout route, it also clears the session.
    - Displays a flash message to inform the user that they have been logged out.
    - Renders the home.html template, likely redirecting the user to the homepage of the application.

Summary

Both logout routes provide a straightforward mechanism for clearing user sessions and providing feedback about the logout process. The general logout route redirects users to the homepage, ensuring a smooth transition after logging out, while the trainer logout route focuses on trainer-specific feedback.