code-documentation

September 21, 2024

Nirmal Sai Swaroop Janapaneedi

1 GrihaSeva Web Application Documentation (Python Flask App)

1.0.1 Overview

This document serves as the technical documentation for the *GrihaSeva* web application, an on-demand home services platform. The application is built using the **Flask** web framework, integrates with a **MySQL** database for storing user and service data, and uses **Flask-Mail** for handling email notifications.

The project is designed to allow users to book a variety of home services, such as cleaning, plumbing, electrical work, and exclusive women-focused services. Service providers can register, manage their profiles, track their earnings, and manage bookings via dedicated interfaces.

This documentation will walk through the various components of the web application, explaining the functionality and design of key modules. The code is broken down into sections, detailing its purpose, the technologies used, and how it works together to provide the intended functionality.

1.1 Libraries Overview

[ ]:
```
from flask import Flask, jsonify, render_template, request, redirect, url_for,␣
↪g, session, flash
import mysql.connector
from decimal import Decimal
import os
import logging
import json
from flask_mail import Mail, Message
import secrets
from datetime import datetime, timedelta
```

Flask: A lightweight WSGI web application framework in Python. It provides tools, libraries, and technologies for building web applications.

jsonify: A helper function from Flask that converts Python dictionaries into JSON responses, making it easy to send data to clients.

render_template: A function used to render HTML templates, allowing for dynamic content generation.

request: An object that contains data sent with HTTP requests, including form data and query parameters.

redirect: A function that generates a response that redirects the client to a different URL.

url_for: A function that generates a URL for a given endpoint, facilitating easy URL management.

g: A special object that allows storing data for the duration of a request, useful for storing data like user information or database connections.

session: A Flask object used to manage user sessions, allowing for storing information across requests.

flash: A function used to send one-time messages to the user, typically for success or error notifications.

mysql.connector: A MySQL driver for Python, allowing you to connect to a MySQL database and execute queries.

Decimal: A built-in Python type for representing decimal numbers with high precision, useful for financial calculations.

os: A module providing functions to interact with the operating system, such as file management and environment variables.

logging: A module for generating log messages, helpful for debugging and tracking application behavior.

json: A module for parsing JSON data and converting Python objects to JSON format.

flask_mail: An extension for Flask that provides easy email sending capabilities. It simplifies tasks like sending notifications and confirmations.

secrets: A module for generating cryptographically strong random numbers suitable for managing data such as passwords or tokens.

datetime: A module for manipulating dates and times, providing classes for date, time, and datetime objects.

timedelta: A class in the datetime module that represents a duration, often used for date arithmetic.

1.2 Application setup & Email configuration

[ ]:
```
## Application setup
app = Flask(__name__)
app.secret_key = os.urandom(24)

## Email configuration
app.config['MAIL_SERVER'] = 'smtp.gmail.com'
app.config['MAIL_PORT'] = 587
app.config['MAIL_USE_TLS'] = True
app.config['MAIL_USERNAME'] = 'nirmaldummy@gmail.com'
```

```
app.config['MAIL_PASSWORD'] = 'ncqx lbqx wbnl yyjj'
mail = Mail(app)
```

Application Setup -

1. app = Flask(**name**): Initializes a new Flask application instance, where **name** refers to the current module. This is necessary for Flask to know where to look for templates, static files, and other resources.
2. app.secret_key = os.urandom(24): Sets a secret key for the application, generated randomly using os.urandom(24). This key is essential for session management and CSRF protection.

Email Configuration -

1. app.config['MAIL_SERVER'] = 'smtp.gmail.com': Configures the mail server to use Gmail's SMTP server.
2. app.config['MAIL_PORT'] = 587: Sets the port for the mail server. Port 587 is used for secure SMTP connections.
3. app.config['MAIL_USE_TLS'] = True: Enables TLS (Transport Layer Security) for the connection to the mail server, ensuring that emails are sent securely.
4. app.config['MAIL_USERNAME'] = 'nirmaldummy@gmail.com': Specifies the email address to use for sending emails.
5. app.config['MAIL_PASSWORD'] = 'ncqx lbqx wbnl yyjj': Sets the password for the email account. (Note: It's recommended to use environment variables or a configuration file to store sensitive information like passwords.)
6. mail = Mail(app): Initializes the Flask-Mail extension with the app configuration, enabling email functionality within the Flask application.

1.3 Database Connection Setup

[ ]:
```
## Establishing database connection setup
def get_db_connection():
    if 'conn' not in g:
        try:
            g.conn = mysql.connector.connect(
                host="127.0.0.1",
                user="root",
                password="Pandu@7463",
                database="mydatabase",
                auth_plugin='mysql_native_password'
            )
            g.cursor = g.conn.cursor(dictionary=True)
        except mysql.connector.Error as err:
            print(f"Error: {err}")
            return None, None
    return g.conn, g.cursor
```

```
@app.teardown_appcontext
def close_db_connection(exception):
    conn = g.pop('conn', None)
    cursor = g.pop('cursor', None)
    if cursor:
        cursor.close()
    if conn:
        conn.close()
```

Database Connection Setup -

get_db_connection() Purpose: Establishes a connection to the MySQL database. It checks if a connection already exists in the Flask g object; if not, it creates a new one.

Flow:

1. Checks if 'conn' is not in g.
2. Tries to connect to the MySQL database using mysql.connector.connect() with the provided connection parameters:
    1. host: The hostname or IP address of the database server (e.g., 127.0.0.1 for localhost).
    2. user: The username to access the database (e.g., root).
    3. password: The password associated with the database user.
    4. database: The name of the database to connect to (e.g., mydatabase).
    5. auth_plugin: Specifies the authentication plugin to use (e.g., mysql_native_password).
3. If successful, a cursor is created with dictionary=True to return query results as dictionaries.
4. If there is an error during connection, it prints the error message and returns None for both connection and cursor.

close_db_connection(exception) Purpose: Closes the database connection and cursor when the application context ends.

Flow:

1. Pops the connection and cursor from g (if they exist).
2. If a cursor is present, it closes it.
3. If a connection is present, it closes it.

Called Automatically: This function is registered as a teardown function, which Flask calls automatically after each request, ensuring that database resources are released properly.

1.4 Application-Related Database Table queries

Note that the following queries are written and executed in MySQL Workbench.

[ ]:
```
'''
CREATE TABLE users (
    id INT AUTO_INCREMENT PRIMARY KEY,
```

```
    username VARCHAR(50) NOT NULL,
    email VARCHAR(100) NOT NULL,
    password VARCHAR(100) NOT NULL
);

CREATE TABLE appointments (
    id INT AUTO_INCREMENT PRIMARY KEY,
    date DATE NOT NULL,
    time TIME NOT NULL,
    service VARCHAR(255) NOT NULL,
    provider VARCHAR(255) NOT NULL,
    notes TEXT,
    username VARCHAR(255) NOT NULL
);

CREATE TABLE password_reset_tokens (
    id INT AUTO_INCREMENT PRIMARY KEY,
    user_id INT NOT NULL,
    token VARCHAR(100) NOT NULL,
    expiry DATETIME NOT NULL,
    FOREIGN KEY (user_id) REFERENCES users(id)
);

CREATE TABLE service_providers (
    id INT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(255) NOT NULL,
    username VARCHAR(255) NOT NULL,
    password VARCHAR(255) NOT NULL,
    email VARCHAR(255) NOT NULL,
    service_type VARCHAR(100) NOT NULL
);

CREATE TABLE provider_details (
    id INT AUTO_INCREMENT PRIMARY KEY,
    provider_username VARCHAR(255),
    name VARCHAR(255),
    phone VARCHAR(20),
    address TEXT,
    FOREIGN KEY (provider_username) REFERENCES service_providers(username)
);

CREATE TABLE provider_earnings (
    id INT AUTO_INCREMENT PRIMARY KEY,
    provider_name VARCHAR(100),
    earnings DECIMAL(10, 2)
);
```

```
CREATE TABLE provider_password_reset_tokens (
    id INT AUTO_INCREMENT PRIMARY KEY,
    provider_id INT,
    token VARCHAR(255),
    expiry DATETIME,
    FOREIGN KEY (provider_id) REFERENCES service_providers(id)
);

CREATE TABLE women_service_providers (
    id INT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(100),
    username VARCHAR(50),
    email VARCHAR(100),
    domain VARCHAR(50),
    phone VARCHAR(50),
    password VARCHAR(255),
    age INT,
    gov_id VARCHAR(50)
);

CREATE TABLE women_service_appointments (
    id INT AUTO_INCREMENT PRIMARY KEY,
    date DATE,
    time TIME,
    service VARCHAR(255),
    provider VARCHAR(255),
    notes TEXT,
    username VARCHAR(255)
);

CREATE TABLE women_provider_earnings (
    id INT AUTO_INCREMENT PRIMARY KEY,
    provider_name VARCHAR(255),
    earnings DECIMAL(10,2)
);

CREATE TABLE women_password_reset_tokens (
    id INT AUTO_INCREMENT PRIMARY KEY,
    provider_id INT,
    token VARCHAR(255),
    expiry DATETIME,
    FOREIGN KEY (provider_id) REFERENCES women_service_providers(id)
);
'''
```

1.5 User-Related Routing

1. Forgot and Reset Password

1.1 forgot_password()

```
[ ]: @app.route('/forgot-password', methods=['GET', 'POST'])
def forgot_password():
    if request.method == 'POST':
        email = request.form['email']
        conn, cursor = get_db_connection()
        if not conn or not cursor:
            flash('Database connection error', 'error')
            return redirect(url_for('forgot_password'))

        try:
            cursor.execute('SELECT id FROM users WHERE email = %s', (email,))
            user = cursor.fetchone()

            if user:
                token = secrets.token_urlsafe(32)
                expiry = datetime.now() + timedelta(hours=1)

                # Insert new token into password_reset_tokens table
                cursor.execute('INSERT INTO password_reset_tokens (user_id,␣
 ↪token, expiry) VALUES (%s, %s, %s)',
                               (user['id'], token, expiry))
                conn.commit()

                reset_link = url_for('reset_password', token=token,␣
 ↪_external=True)
                send_reset_email(email, reset_link)

                flash('Password reset link has been sent to your email.',␣
 ↪'success')
                return redirect(url_for('user_login'))
            else:
                flash('Email not found. Please check and try again.', 'error')
        except mysql.connector.Error as err:
            flash(f'An error occurred: {err}', 'error')
        finally:
            cursor.close()
            conn.close()

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

1. URL: /forgot-password
2. Methods: GET, POST

3. Description: Handles password reset requests by sending a reset link to the user's email.
4. Functionality:
    1. Accepts email input.
    2. Validates if the email exists in the database.
    3. Generates a reset token and expiry time.
    4. Sends a reset email with the link.
5. Returns: Redirects to login or the same page with appropriate messages.

1.2 reset_password()

[ ]:
```
def send_reset_email(email, reset_link):
    msg = Message('Password Reset Request',
                  sender='noreply@yourdomain.com',
                  recipients=[email])
    msg.body = f'''To reset your password, visit the following link:
{reset_link}

If you did not make this request then simply ignore this email and no changes␣
 ↪will be made.
'''
    mail.send(msg)

@app.route('/reset-password/<token>', methods=['GET', 'POST'])
def reset_password(token):
    conn, cursor = get_db_connection()
    if not conn or not cursor:
        flash('Database connection error', 'error')
        return redirect(url_for('user_login'))

    try:
        # Join password_reset_tokens with users table to get user information
        cursor.execute('''
            SELECT u.id, u.email
            FROM password_reset_tokens t
            JOIN users u ON t.user_id = u.id
            WHERE t.token = %s AND t.expiry > NOW()
        ''', (token,))
        result = cursor.fetchone()

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

                if new_password == confirm_password:
                    # Update user's password
```

```
                    cursor.execute('UPDATE users SET password = %s WHERE id =␣
↪%s',
                                   (new_password, result['id']))
                    # Delete the used token
                    cursor.execute('DELETE FROM password_reset_tokens WHERE␣
↪token = %s', (token,))
                    conn.commit()
                    flash('Your password has been updated successfully.',␣
↪'success')
                    return redirect(url_for('user_login'))
                else:
                    flash('Passwords do not match. Please try again.', 'error')

            return render_template('reset_password.html', token=token)
        else:
            flash('Invalid or expired reset token. Please try again.', 'error')
            return redirect(url_for('forgot_password'))
    except mysql.connector.Error as err:
        flash(f'An error occurred: {err}', 'error')
    finally:
        cursor.close()
        conn.close()

    return redirect(url_for('user_login'))
```

1. URL: /reset-password/
2. Methods: GET, POST
3. Description: Resets the user's password using the provided token.
4. Functionality:
    1. Validates the token and checks its expiry.
    2. Accepts new password input.
    3. Updates the user's password if valid.
    4. Deletes the used token from the database.
5. Returns: Redirects to login or the same page with messages.

2. User Home and Profile Pages

2.1 user_home()

[ ]:
```
@app.route('/user-home')
def user_home():
    if 'user_id' not in session:
        return redirect(url_for('user_login'))
    return render_template('user_home.html')
```

1. URL: /user-home
2. Methods: GET
3. Description: Displays the user home page after login.
4. Returns: Renders the user home template.

2.2 user_profile()

```
[ ]: @app.route('/user-profile')
def user_profile():
    user = session.get('user')
    return render_template('user_profile.html', user = user)
```

1. URL: /user-profile
2. Methods: GET
3. Description: Displays the user's profile information.
4. Returns: Renders the user profile template with user data.

3. User Login Routing

3.1 user_login()

```
[ ]: @app.route('/user-login', methods=['GET', 'POST'])
def user_login():
    error = None
    if request.method == 'POST':
        username = request.form.get('username')
        password = request.form.get('password')

        conn, cursor = get_db_connection()
        if not conn or not cursor:
            return "Database connection error", 500

        cursor.execute(
            "SELECT * FROM users WHERE username = %s AND password = %s",
            (username, password)
        )
        user = cursor.fetchone()

        if user:
            session['user_id'] = user['id']
            session['username'] = user['username']
            return redirect(url_for('user_home'))
        else:
            error = "Invalid username or password"
    return render_template('user_login.html', error=error)
```

1. URL: /user-login
2. Methods: GET, POST
3. Description: Authenticates the user and initiates a session.
4. Functionality:
    1. Accepts username and password.
    2. Validates against the database.
    3. Starts a user session upon successful login.
5. Returns: Redirects to user home or displays an error message.

4. User Registration Routing

4.1 user_register()

[ ]:
```
@app.route('/user-register', methods=['GET', 'POST'])
def user_register():
    if request.method == 'POST':
        username = request.form.get('username')
        email = request.form.get('email')
        password = request.form.get('password')

        conn, cursor = get_db_connection()
        if not conn or not cursor:
            return "Database connection error", 500

        try:
            cursor.execute(
                "INSERT INTO users (username, email, password) VALUES (%s, %s,␣
↪%s)",
                (username, email, password)
            )
            conn.commit()
            return redirect(url_for('user_login'))
        except mysql.connector.IntegrityError:
            return render_template('user_registration.html', error="Username or␣
↪email already exists")
    return render_template('user_registration.html')
```

1. URL: /user-register
2. Methods: GET, POST
3. Description: Registers a new user in the system.
4. Functionality:
    1. Accepts username, email, and password.
    2. Checks for existing username/email.

3. Inserts new user into the database.

5. Returns: Redirects to login or displays an error message.

5. User Profile and Password Update

5.1 profile()

[ ]:
```
@app.route('/profile', methods=['GET', 'POST'])
def profile():
    if 'username' not in session:
        return redirect(url_for('login'))

    conn = get_db_connection()
    user = conn.execute('SELECT * FROM users WHERE username = ?',␣
 ↪(session['username'],)).fetchone()

    if request.method == 'POST':
        name = request.form['name']
        phone = request.form['phone']
        address = request.form['address']

        conn.execute('UPDATE users SET name = ?, phone = ?, address = ? WHERE␣
 ↪username = ?',
                     (name, phone, address, session['username']))
        conn.commit()
        flash('Profile updated successfully!', 'success')

        conn.close()
        return redirect(url_for('profile'))
    return render_template('profile.html', user=user)
```

1. URL: /profile
2. Methods: GET, POST
3. Description: Displays and updates user profile information.
4. Functionality:
    1. Accepts and updates user details (name, phone, address).
5. Returns: Redirects to the profile page with success message.

5.2 update_password()

[ ]:
```
@app.route('/update_password', methods=['GET', 'POST'])
def update_password():
    if 'username' not in session:
        flash('You must be logged in to change your password.', 'error')
        return redirect(url_for('login'))
```

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

      if not all([current_password, new_password, confirm_password]):
          flash('All fields are required.', 'error')
          return redirect(url_for('update_password'))

      if new_password != confirm_password:
          flash('New passwords do not match. Please try again.', 'error')
          return redirect(url_for('update_password'))

      conn, cursor = get_db_connection()
      if not conn or not cursor:
          flash('Database connection error', 'error')
          return redirect(url_for('update_password'))

      try:
          cursor.execute('SELECT * FROM users WHERE username = %s',␣
↪(session['username'],))
          user = cursor.fetchone()

          if user and current_password == user['password']:
              cursor.execute('UPDATE users SET password = %s WHERE username =␣
↪%s',
                             (new_password, session['username']))
              conn.commit()
              flash('Password updated successfully!', 'success')
          else:
              flash('Current password is incorrect. Please try again.',␣
↪'error')
      except mysql.connector.Error as err:
          flash(f'An error occurred: {err}', 'error')
      finally:
          cursor.close()
          conn.close()
  return render_template('user_profile.html', user = user)
```

1. URL: /update_password
2. Methods: GET, POST
3. Description: Allows users to change their password.
4. Functionality:
    1. Validates current password and new password.

2. Updates the password in the database.

5. Returns: Redirects to the profile page with success/error messages.

6.Appointment Booking Functionality

6.1 book_appointment()

[ ]:
```python
@app.route('/book-appointment', methods=['GET', 'POST'])
def book_appointment():
    if 'username' not in session:
        flash('Please log in to book an appointment.', 'error')
        return redirect(url_for('user_login'))

    if request.method == 'POST':
        # Handle form submission and book the appointment
        date = request.form['date']
        time = request.form['time']
        service = request.form['service']
        provider = request.form.get('service-provider')
        notes = request.form['notes']
        username = session['username']  # Get the username from the session

        if not provider:
            flash('Please select a service provider.', 'error')
            return redirect(url_for('book_appointment', service=service))

        # Insert the appointment details into the database
        conn, cursor = get_db_connection()
        if not conn or not cursor:
            return "Database connection error", 500

        try:
            cursor.execute(
                "INSERT INTO appointments (date, time, service, provider,␣
↪notes, username) VALUES (%s, %s, %s, %s, %s, %s)",
                (date, time, service, provider, notes, username)
            )
            conn.commit()
            flash('Appointment booked successfully!', 'success')
            return redirect(url_for('user_home'))
        except mysql.connector.Error as err:
            flash(f'An error occurred: {err}', 'error')
            return redirect(url_for('book_appointment', service=service))
        finally:
            cursor.close()
            conn.close()
```

```
    else:
        # Retrieve the list of available services
        services = get_available_services()

        # Retrieve the list of service providers based on the selected service
        selected_service = request.args.get('service')
        service_providers = get_service_providers(selected_service) if␣
↪selected_service else []
        return render_template('user_book_appointment.html', services=services,␣
↪service_providers=service_providers, selected_service=selected_service)
```

[ ]:
```
def get_available_services():
    conn, cursor = get_db_connection()
    if not conn or not cursor:
        return []

    try:
        cursor.execute("SELECT DISTINCT service_type FROM service_providers")
        services = [row['service_type'] for row in cursor.fetchall()]
        return services
    except mysql.connector.Error as err:
        flash(f'An error occurred: {err}', 'error')
        return []
    finally:
        cursor.close()
        conn.close()

def get_service_providers(service):
    conn, cursor = get_db_connection()
    if not conn or not cursor:
        return []

    try:
        cursor.execute(
            "SELECT name FROM service_providers WHERE service_type = %s",
            (service,)
        )
        service_providers = cursor.fetchall()
        return [provider['name'] for provider in service_providers]
    except mysql.connector.Error as err:
        flash(f'An error occurred: {err}', 'error')
        return []
    finally:
        cursor.close()
        conn.close()

@app.route('/get-providers/<service>', methods=['GET'])
```

```
def get_providers(service):
    providers = get_service_providers(service)
    return jsonify(providers)

def get_service_providers(service):
    conn, cursor = get_db_connection()
    if not conn or not cursor:
        return []

    try:
        cursor.execute(
            "SELECT name FROM service_providers WHERE service_type = %s",
            (service,)
        )
        return [row['name'] for row in cursor.fetchall()]
    except mysql.connector.Error as err:
        flash(f'An error occurred: {err}', 'error')
        return []
    finally:
        cursor.close()
        conn.close()
```

1. URL: /book-appointment
2. Methods: GET, POST
3. Description: Allows users to book appointments.
4. Functionality:
    1. Accepts appointment details (date, time, service, provider, notes).
    2. Inserts appointment into the database.
5. Returns: Redirects to user home or displays an error message.

6.2 book_appointment_women_services()

[ ]:
```
@app.route('/book-appointment-women-services', methods=['GET', 'POST'])
def book_appointment_women_services():
    if 'username' not in session:
        flash('Please log in to book an appointment.', 'error')
        return redirect(url_for('user_login'))

    if request.method == 'POST':
        # Handle form submission and book the appointment
        date = request.form['date']
        time = request.form['time']
        service = request.form['service']
        provider = request.form.get('women-service-provider')
        notes = request.form['notes']
```

```
        username = session['username']  # Get the username from the session

        if not provider:
            flash('Please select a service provider.', 'error')
            return redirect(url_for('book_appointment_women_services',␣
↪service=service))

        # Insert the appointment details into the database
        conn, cursor = get_db_connection()
        if not conn or not cursor:
            return "Database connection error", 500

        try:
            cursor.execute(
                "INSERT INTO women_service_appointments (date, time, service,␣
↪provider, notes, username) VALUES (%s, %s, %s, %s, %s, %s)",
                (date, time, service, provider, notes, username)
            )
            conn.commit()
            flash('Women services appointment booked successfully!', 'success')
            return redirect(url_for('user_home'))
        except mysql.connector.Error as err:
            flash(f'An error occurred: {err}', 'error')
            return redirect(url_for('book_appointment_women_services',␣
↪service=service))
        finally:
            cursor.close()
            conn.close()

    else:
        # Retrieve the list of available women's services
        services = get_available_women_services()

        # Retrieve the list of women service providers based on the selected␣
↪service
        selected_service = request.args.get('service')
        service_providers = get_women_service_providers(selected_service) if␣
↪selected_service else []

        return render_template('user_book_appointmentw.html',␣
↪services=services, service_providers=service_providers,␣
↪selected_service=selected_service)

def get_available_women_services():
    conn, cursor = get_db_connection()
    if not conn or not cursor:
```

```
        return []

    try:
        cursor.execute("SELECT DISTINCT domain FROM women_service_providers")
        services = [row['domain'] for row in cursor.fetchall()]
        return services
    except mysql.connector.Error as err:
        flash(f'An error occurred: {err}', 'error')
        return []
    finally:
        cursor.close()
        conn.close()

def get_women_service_providers(service):
    conn, cursor = get_db_connection()
    if not conn or not cursor:
        return []

    try:
        cursor.execute(
            "SELECT name, age FROM women_service_providers WHERE domain = %s",
            (service,)
        )
        return [{'name': row['name'], 'age': row['age']} for row in cursor.
 ↪fetchall()]
    except mysql.connector.Error as err:
        flash(f'An error occurred: {err}', 'error')
        return []
    finally:
        cursor.close()
        conn.close()

@app.route('/get-women-service-providers/<service>', methods=['GET'])
def get_women_providers(service):
    providers = get_women_service_providers(service)
    return jsonify(providers)
```

1. URL: /book-appointment-women-services
2. Methods: GET, POST
3. Description: Allows users to book women's exclusive services appointments.
4. Functionality: Similar to book_appointment but specifically for women's services.
5. Returns: Redirects to user home or displays an error message.

7. User Bookings Routing

7.1 my_bookings()

[ ]:
```python
@app.route('/my-bookings')
def my_bookings():
    return render_template('my_bookings.html')
```

1. URL: /my-bookings
2. Methods: GET
3. Description: Displays the user's booking history.
4. Returns: Renders the my bookings template.

7.2 api_my_bookings()

[ ]:
```python
@app.route('/api/my-bookings')
def api_my_bookings():
    logging.debug("Entering api_my_bookings function")
    conn, cursor = None, None
    try:
        conn, cursor = get_db_connection()
        if not conn or not cursor:
            logging.error("Failed to establish a database connection.")
            return jsonify({"error": "Database connection error"}), 500

        username = session.get('username')
        if not username:
            logging.warning("User is not logged in.")
            return jsonify({"error": "User not logged in"}), 401

        logging.debug(f"Username passed to query: {username}")

        # Query for regular appointments
        regular_query = """
            SELECT service, provider AS name, date, time, 'regular' AS␣
↪booking_type
            FROM appointments
            WHERE username = %s
        """
        cursor.execute(regular_query, (username,))
        regular_bookings = cursor.fetchall()

        # Query for women's services appointments
        women_query = """
            SELECT service, provider AS name, date, time, 'women_services' AS␣
↪booking_type
            FROM women_service_appointments
            WHERE username = %s
        """
```

```
        cursor.execute(women_query, (username,))
        women_bookings = cursor.fetchall()

        # Combine and sort all bookings
        all_bookings = regular_bookings + women_bookings
        all_bookings.sort(key=lambda x: (x['date'], x['time']), reverse=True)

        if all_bookings:
            bookings = []
            for row in all_bookings:
                bookings.append({
                    "service": row["service"],
                    "name": row["name"],
                    "date": row["date"].strftime('%Y-%m-%d'),
                    "time": str(row["time"]),
                    "booking_type": row["booking_type"]
                })
            logging.debug(f"Bookings fetched: {bookings}")
            return jsonify({"bookings": bookings})
        else:
            logging.info("No bookings found for the user.")
            return jsonify({"message": "No bookings found"}), 200
    except Exception as e:
        logging.exception("An unexpected error occurred.")
        return jsonify({"error": str(e)}), 500
    finally:
        if cursor:
            cursor.close()
        if conn:
            conn.close()
        logging.debug("Exiting api_my_bookings function")
```

1. URL: /api/my-bookings
2. Methods: GET
3. Description: API endpoint to fetch user bookings.
4. Returns: JSON response containing the user's bookings.

7.3 cancel_booking()

[ ]:
```
@app.route('/api/cancel-booking', methods=['POST'])
def cancel_booking():
    logging.debug("Entering cancel_booking function")
    conn, cursor = None, None
    try:
        conn, cursor = get_db_connection()
        if not conn or not cursor:
```

```
            logging.error("Failed to establish a database connection.")
            return jsonify({"error": "Database connection error"}), 500

        data = request.get_json()
        date = data.get('date')
        time = data.get('time')
        booking_type = data.get('booking_type')
        username = session.get('username')

        if not username:
            logging.warning("User is not logged in.")
            return jsonify({"error": "User not logged in"}), 401

        logging.debug(f"Cancelling {booking_type} booking for user: {username}␣
↪on {date} at {time}")

        if booking_type == 'regular':
            table = 'appointments'
        elif booking_type == 'women_services':
            table = 'women_services_appointments'
        else:
            return jsonify({"error": "Invalid booking type"}), 400

        query = f"""
            DELETE FROM {table}
            WHERE username = %s AND date = %s AND time = %s
        """
        cursor.execute(query, (username, date, time))
        conn.commit()

        if cursor.rowcount > 0:
            logging.info(f"{booking_type.capitalize()} booking cancelled for␣
↪user: {username}")
            return jsonify({"success": True}), 200
        else:
            logging.warning("No matching booking found to cancel.")
            return jsonify({"success": False, "message": "No matching booking␣
↪found"}), 404
    except Exception as e:
        logging.exception("An unexpected error occurred.")
        return jsonify({"error": str(e)}), 500
    finally:
        if cursor:
            cursor.close()
        if conn:
            conn.close()
        logging.debug("Exiting cancel_booking function")
```

1. URL: /api/cancel-booking
2. Methods: POST
3. Description: Cancels a user's booking.
4. Functionality:
    1. Accepts booking details and deletes the booking from the database.
5. Returns: JSON response indicating success or failure.

1.6 General Routings Documentation

1. Home Page

```
[ ]: @app.route('/')
     def home():
         return render_template('index.html')
```

URL: /

Methods: GET

Description: Displays the home page of the web application.

Returns: Renders the index.html template.

2. Services Page

```
[ ]: @app.route('/services')
     def services():
         return render_template('services.html')
```

URL: /services

Methods: GET

Description: Displays the services page listing available services.

Returns: Renders the services.html template.

3. About Us Page

```
[ ]: @app.route('/about')
     def about():
         return render_template('about.html')
```

URL: /about

Methods: GET

Description: Displays the about page with information about the company.

Returns: Renders the about.html template.

4. Contact Page

```
[ ]: @app.route('/contact')
     def contact():
         return render_template('contact.html')
```

URL: /contact

Methods: GET

Description: Displays the contact page with contact details and form.

Returns: Renders the contact.html template.

5. Private Policy Page

```
[ ]: @app.route('/privacy')
     def privacy():
         return render_template('privacy.html')
```

URL: /privacy

Methods: GET

Description: Displays the privacy policy of the application.

Returns: Renders the privacy.html template.

6. Terms of Service Page

```
[ ]: @app.route('/tos')
     def tos():
         return render_template('termsandservices.html')
```

URL: /tos

Methods: GET

Description: Displays the terms of service (TOS) page.

Returns: Renders the termsandservices.html template.

7. Deciding Page

```
[ ]: @app.route('/deciding')
     def deciding():
         return render_template('deciding.html')
```

URL: /deciding

Methods: GET

Description: Displays a custom page (e.g., “deciding” page with decision-making features).

Returns: Renders the deciding.html template.

8. Main Blogs Page and Individual Blog Pages

[ ]:
```python
@app.route('/blogs')
def blogs():
    return render_template('blogs.html')

@app.route('/blog1')
def blog1():
    return render_template('blog1.html')

@app.route('/blog2')
def blog2():
    return render_template('blog2.html')

@app.route('/blog3')
def blog3():
    return render_template('blog3.html')

@app.route('/blog4')
def blog4():
    return render_template('blog4.html')

@app.route('/blog5')
def blog5():
    return render_template('blog5.html')

@app.route('/blog6')
def blog6():
    return render_template('blog6.html')

@app.route('/blog7')
def blog7():
    return render_template('blog7.html')

@app.route('/blog8')
def blog8():
    return render_template('blog8.html')
```

8.0

URL: /blogs

Methods: GET

Description: Displays a list of all blogs.

Returns: Renders the blogs.html template.

8.1 blog1

URL: /blog1

Methods: GET

Description: Displays the first blog.

Returns: Renders the blog1.html template.

8.2 blog2

URL: /blog2

Methods: GET

Description: Displays the second blog.

Returns: Renders the blog2.html template.

8.3. blog3

URL: /blog3

Methods: GET

Description: Displays the third blog.

Returns: Renders the blog3.html template.

8.4. blog4

URL: /blog4

Methods: GET

Description: Displays the fourth blog.

Returns: Renders the blog4.html template.

8.5. blog5

URL: /blog5

Methods: GET

Description: Displays the fifth blog.

Returns: Renders the blog5.html template.

8.6. blog6

URL: /blog6

Methods: GET

Description: Displays the sixth blog.

Returns: Renders the blog6.html template.

8.7. blog7

URL: /blog7

Methods: GET

Description: Displays the seventh blog.

Returns: Renders the blog7.html template.

8.8. blog8

URL: /blog8

Methods: GET

Description: Displays the eighth blog.

Returns: Renders the blog8.html template.

9. Women Exclusive Services Page

```
[ ]: @app.route('/women')
     def women():
         return render_template('women.html')
```

URL: /women

Methods: GET

Description: Displays the women-exclusive services page.

Returns: Renders the women.html template.

10. Logout

```
[ ]: @app.route('/logout')
     def logout():
         session.pop('user_id', None)
         session.pop('provider_id', None)
         session.pop('username', None)
         return redirect(url_for('home'))
```

URL: /logout

Methods: GET

Description: Logs out the user or provider by clearing session data.

Functionality: Clears user_id, provider_id, and username from the session.

Returns: Redirects to the home page.

1.7 Service Provider-Related Routing

1. Provider Login and Registration

1.1 provider_login()

```
[ ]: @app.route('/provider-login', methods=['GET', 'POST'])
     def provider_login():
         error = None
         if request.method == 'POST':
             username = request.form.get('username')
             password = request.form.get('password')
```

```
        conn, cursor = get_db_connection()
        if not conn or not cursor:
            return "Database connection error", 500

        cursor.execute(
            "SELECT * FROM service_providers WHERE username = %s AND password =␣
↪%s",
            (username, password)
        )
        provider = cursor.fetchone()

        if provider:
            session['provider_id'] = provider['id']
            session['username'] = provider['username']
            return redirect(url_for('provider_home'))
        else:
            error = "Invalid username or password"
    return render_template('provider_login.html', error=error)
```

URL: /provider-login

Methods: GET, POST

Description: Authenticates the service provider and initiates a session.

Functionality:

1. Accepts username and password from the form.

2. Validates against the service_providers database.

3. If the credentials are valid, it starts a session and stores the provider_id and username.

4. If the credentials are invalid, an error message is displayed.

Returns: Redirects to the provider home page or re-displays the login page with an error message.

1.2 provider_register()

[ ]:
```
@app.route('/provider-register', methods=['GET', 'POST'])
def provider_register():
    if request.method == 'POST':
        name = request.form.get('name')
        username = request.form.get('username')
        password = request.form.get('password')
        email = request.form.get('email')
        service_type = request.form.get('service_type')

        conn, cursor = get_db_connection()
```

```
        if not conn or not cursor:
            return "Database connection error", 500

        try:
            cursor.execute(
                "INSERT INTO service_providers (name, username, password,␣
↪email, service_type) VALUES (%s, %s, %s, %s, %s)",
                (name, username, password, email, service_type)
            )
            conn.commit()
            return redirect(url_for('provider_login'))
        except mysql.connector.IntegrityError:
            return render_template('provider_registration.html',␣
↪error="Username or email already exists")
    return render_template('provider_registration.html')
```

URL: /provider-register

Methods: GET, POST

Description: Registers a new service provider in the system.

Functionality:

1. Accepts details like name, username, password, email, and service_type.

2. Checks for existing username or email in the service_providers database.

3. Inserts the new provider into the database if the inputs are valid.

Returns: Redirects to the login page or shows a registration error.

2. Provider Home, Profile Pages & Password updating

2.1 provider_home()

[ ]:
```
@app.route('/provider-home')
def provider_home():
    if 'provider_id' not in session:
        return redirect(url_for('provider_login'))
    return render_template('provider_home.html')
```

URL: /provider-home

Methods: GET

Description: Displays the service provider's home page after login.

Functionality:

1. Checks if the provider_id is in the session to ensure the provider is logged in.

2. If the provider is not logged in, it redirects to the login page.

3. If logged in, renders the provider home page.

Returns: Renders the provider_home.html template or redirects to the login page.

2.2 provider_profile()

[ ]:
```
@app.route('/provider-profile', methods=['GET', 'POST'])
def provider_profile():
    if 'username' not in session:
        flash('Please log in to access your profile.', 'error')
        return redirect(url_for('provider_login'))

    conn, cursor = get_db_connection()
    if not conn or not cursor:
        flash('Database connection error. Please try again later.', 'error')
        return render_template('provider_profile.html')
    provider = None
    try:
        # Fetch provider details
        cursor.execute('SELECT * FROM provider_details WHERE provider_username␣
↪= %s', (session['username'],))
        provider = cursor.fetchone()

        if request.method == 'POST':
            name = request.form.get('name')
            phone = request.form.get('phone')
            address = request.form.get('address')

            if provider:
                # Update existing details
                cursor.execute('UPDATE provider_details SET name = %s, phone =␣
↪%s, address = %s WHERE provider_username = %s',
                               (name, phone, address, session['username']))
            else:
                # Insert new details
                cursor.execute('INSERT INTO provider_details␣
↪(provider_username, name, phone, address) VALUES (%s, %s, %s, %s)',
                               (session['username'], name, phone, address))

            conn.commit()
            flash('Profile updated successfully!', 'success')

            # Fetch updated details
            cursor.execute('SELECT * FROM provider_details WHERE␣
↪provider_username = %s', (session['username'],))
            provider = cursor.fetchone()
```

```
    except mysql.connector.Error as err:
        conn.rollback()
        flash(f'An error occurred: {err}', 'error')
    finally:
        cursor.close()
        conn.close()
    return render_template('provider_profile.html', provider=provider)
```

URL: /provider-profile

Methods: GET

Description: Displays the service provider's profile information.

Functionality:

1. Fetches provider data from the service_providers database based on the provider_id in the se

2. Displays profile information such as name, email, and service_type.

Returns: Renders the provider_profile.html template with provider data.

2.3 Provider password updating

[ ]:
```
@app.route('/provider_update_password', methods=['GET', 'POST'])
def provider_update_password():
    if 'username' not in session:
        flash('You must be logged in to change your password.', 'error')
        return redirect(url_for('provider_login'))

    provider = None
    if request.method == 'POST':
        current_password = request.form.get('current_password')
        new_password = request.form.get('new_password')
        confirm_password = request.form.get('confirm_password')

        if not all([current_password, new_password, confirm_password]):
            flash('All fields are required.', 'error')
            return redirect(url_for('provider_update_password'))

        if new_password != confirm_password:
            flash('New passwords do not match. Please try again.', 'error')
            return redirect(url_for('provider_update_password'))

        conn, cursor = get_db_connection()
        if not conn or not cursor:
            flash('Database connection error', 'error')
            return redirect(url_for('provider_update_password'))
```

```
        try:
            cursor.execute('SELECT * FROM service_providers WHERE username =␣
↪%s', (session['username'],))
            provider = cursor.fetchone()

            if provider:
                stored_password = provider['password']
                if current_password == stored_password:
                    cursor.execute('UPDATE service_providers SET password = %s␣
↪WHERE username = %s',
                                   (new_password, session['username']))
                    conn.commit()
                    flash('Password updated successfully!', 'success')
                else:
                    flash('Current password is incorrect. Please try again.',␣
↪'error')
            else:
                flash('Provider not found.', 'error')

        except mysql.connector.Error as err:
            conn.rollback()
            flash(f'An error occurred: {err}', 'error')
        finally:
            if cursor:
                cursor.close()
            if conn:
                conn.close()
    return render_template('provider_profile.html', provider=provider)
```

URL: /provider_update_password

Methods: GET, POST

Description: Handles updating the service provider's password.

Functionality:

1. Checks if the provider is logged in by verifying if 'username' exists in the session.

2. Displays the password update form if the request method is GET.

3. On POST request:

    3.1. Accepts current_password, new_password, and confirm_password from the form.

    3.2. Verifies if all fields are filled and that the new passwords match.

    3.3. Connects to the database and validates if the current password is correct for the logg

3.4. Updates the provider's password if the current password matches.

3.5. Provides feedback via flash messages based on success or failure.

Returns:

1. If successful, flashes a success message and remains on the same page.

2. If there is an error (incorrect password, database connection issues), flashes an error mes

3. Provider Bookings

3.1 provider_bookings()

[ ]:
```python
@app.route('/provider-bookings')
def provider_bookings():
    if 'username' not in session:
        return redirect(url_for('login'))
    return render_template('provider_bookings.html')
```

URL: /provider-bookings

Methods: GET

Description: Displays the service provider's bookings page.

Functionality:

1. Checks if the provider is logged in ('username' exists in the session).

2. If logged in, renders the provider_bookings.html template where booking information will be

Returns:

1. Renders the provider_bookings.html page if the provider is logged in.

2. Redirects to the login page if not logged in.

3.2 api_provider_bookings()

[ ]:
```python
@app.route('/api/provider-bookings')
def api_provider_bookings():
    logging.info("Entering api_provider_bookings function")
    conn, cursor = None, None
    try:
        # Log the entire session data to ensure the username is set
        logging.info(f"Session data: {session.items()}")

        conn, cursor = get_db_connection()
        if not conn or not cursor:
            logging.error("Failed to establish a database connection.")
            return jsonify({"error": "Database connection error"}), 500
```

```
        username = session.get('username')
        logging.info(f"Provider username from session: {username}")
        if not username:
            logging.warning("User is not logged in.")
            return jsonify({"error": "User not logged in"}), 401

        # Check if the provider exists in the service_providers table
        check_provider_query = "SELECT * FROM service_providers WHERE username␣
↪= %s"
        cursor.execute(check_provider_query, (username,))
        provider = cursor.fetchone()

        if not provider:
            logging.warning(f"No provider found with username: {username}")
            return jsonify({"error": "Provider not found"}), 404

        logging.info(f"Provider found: {provider}")

        # Fetch the bookings for the provider
        query = """
            SELECT * FROM appointments
            WHERE provider = %s
            ORDER BY date DESC, time DESC
        """
        cursor.execute(query, (provider['name'],))
        rows = cursor.fetchall()

        logging.info(f"Query executed. Number of rows returned: {len(rows)}")

        if rows:
            bookings = []
            for row in rows:
                # Log the raw row data to see exactly what is being fetched
                logging.info(f"Raw row data: {row}")

                # Ensure that the client's username is correctly extracted
                client_username = row.get('username', 'Unknown Client')
                if client_username == 'Unknown Client':
                    logging.warning(f"Client username missing for booking ID:␣
↪{row['id']}")

                # Build the booking dictionary
                booking = {
                    "id": row['id'],
                    "date": row['date'].strftime('%Y-%m-%d'),
                    "time": str(row['time']),
```

```
                    "service": row['service'],
                    "provider": row['provider'],
                    "notes": row.get('notes', ''),
                    "client_username": client_username
                }
                bookings.append(booking)
                logging.info(f"Processed booking: {json.dumps(booking)}")

            logging.info(f"Bookings fetched for provider: {json.dumps(bookings,␣
↪indent=2)}")
            return jsonify({"bookings": bookings})
        else:
            logging.info(f"No bookings found for the provider: {username}")
            return jsonify({"message": "No bookings found"}), 200
    except Exception as e:
        logging.exception(f"An unexpected error occurred in provider bookings:␣
↪{str(e)}")
        return jsonify({"error": str(e)}), 500
    finally:
        if cursor:
            cursor.close()
        if conn:
            conn.close()
        logging.info("Exiting api_provider_bookings function")
```

URL: /api/provider-bookings

Methods: GET

Description: API to fetch bookings for the logged-in service provider.

Functionality:

1. Logs all session data for debugging purposes to ensure 'username' is set.

2. Connects to the database and retrieves the provider's details using the session's username.

3. Checks if the provider exists in the service_providers table.

4. Fetches the provider's bookings from the appointments table ordered by date and time.

5. Constructs a list of booking dictionaries, each containing details such as date, time, servi

Returns:

1. If successful, returns a JSON object containing the bookings.

2. If no bookings are found, returns a message indicating "No bookings found".

3. In case of errors (e.g., database connection failure or user not logged in), returns an appr

4. Provider Earnings Page

[ ]:
```
@app.route('/provider-earnings', methods=['GET', 'POST'])
def display_earnings():
    conn, cursor = get_db_connection()
    username = session.get('username')  # Get provider username from session

    if not username:
        print("Provider name is not set in session.")
        return "Error: Provider name is not set in session."

    # Handle POST request to calculate and add earnings
    if request.method == 'POST':
        try:
            hourly_charge = float(request.form['hourlyCharge'])  # Get hourly␣
↪charge
            hours_worked = float(request.form['hoursWorked'])  # Get hours␣
↪worked
            session_earnings = Decimal(hourly_charge * hours_worked)  #␣
↪Calculate session earnings

            # Check if provider already has an entry
            query = "SELECT earnings FROM provider_earnings WHERE provider_name␣
↪= %s"
            cursor.execute(query, (username,))
            result = cursor.fetchone()

            if result:
                # Convert existing earnings to Decimal and update
                existing_earnings = Decimal(result['earnings'])
                total_earnings = existing_earnings + session_earnings
                update_query = "UPDATE provider_earnings SET earnings = %s␣
↪WHERE provider_name = %s"
                cursor.execute(update_query, (total_earnings, username))
            else:
                # Insert new entry for the provider
                total_earnings = session_earnings
                insert_query = "INSERT INTO provider_earnings (provider_name,␣
↪earnings) VALUES (%s, %s)"
                cursor.execute(insert_query, (username, total_earnings))

            conn.commit()

            # Redirect to display updated earnings
            return redirect(url_for('display_earnings'))

        except Exception as e:
```

```
            print(f"An error occurred: {e}")
            return "Error processing your request.", 500

    # Handle GET request to display the earnings page
    query = "SELECT earnings FROM provider_earnings WHERE provider_name = %s"
    cursor.execute(query, (username,))
    result = cursor.fetchone()

    if result and 'earnings' in result:
        total_earnings = Decimal(result['earnings'])
    else:
        total_earnings = Decimal('0.00')  # Default if no record is found

    # Render the earnings page with total earnings
    return render_template('provider_earnings.html',␣
↪total_earnings=total_earnings)
```

URL: /provider-earnings

Methods: GET

Description: Displays the total earnings of the logged-in service provider.

Functionality:

1.Retrieves the total earnings for the provider from the provider_earnings table using provide

2. Dynamically updates and displays the earnings on the page.

Returns: Renders the provider_earnings.html template with earnings data.

5. Provider Password Management Routing

5.1 Provider Forgot Password

[ ]:
```
@app.route('/provider-forgot-password', methods=['GET', 'POST'])
def provider_forgot_password():
    if request.method == 'POST':
        email = request.form['email']
        conn, cursor = get_db_connection()
        if not conn or not cursor:
            flash('Database connection error', 'error')
            return redirect(url_for('provider_forgot_password'))

        try:
            cursor.execute('SELECT id FROM service_providers WHERE email = %s',␣
↪(email,))
            provider = cursor.fetchone()

            if provider:
```

```
                token = secrets.token_urlsafe(32)
                expiry = datetime.now() + timedelta(hours=1)

                # Insert new token into provider_password_reset_tokens table
                cursor.execute('INSERT INTO provider_password_reset_tokens␣
 ↪(provider_id, token, expiry) VALUES (%s, %s, %s)',
                               (provider['id'], token, expiry))
                conn.commit()

                reset_link = url_for('provider_reset_password', token=token,␣
 ↪_external=True)
                send_provider_reset_email(email, reset_link)

                flash('Password reset link has been sent to your email.',␣
 ↪'success')
                return redirect(url_for('provider_login'))
            else:
                flash('Email not found. Please check and try again.', 'error')
        except mysql.connector.Error as err:
            flash(f'An error occurred: {err}', 'error')
        finally:
            cursor.close()
            conn.close()

    return render_template('provider_forgot_password.html')

def send_provider_reset_email(email, reset_link):
    msg = Message('Service Provider Password Reset Request',
                  sender='nirmaldummy@gmail.com',
                  recipients=[email])
    msg.body = f'''To reset your service provider password, visit the following␣
 ↪link:
{reset_link}

If you did not make this request then simply ignore this email and no changes␣
 ↪will be made.
'''
    mail.send(msg)
```

Route: /provider-forgot-password

Methods: GET, POST

Description: Allows service providers to request a password reset link, sent to their registered email.

Functionality:

1. Accepts provider's email address.

2. Validates whether the email exists in the database.

3. Generates a secure token with an expiry time.

4. Stores the token in the provider_password_reset_tokens table.

5. Sends a password reset email with a link to the provider.

Returns:

1. On success: A success message and redirect to the provider login page.

2. On failure: Error messages displayed on the same page.

5.2 Provider Reset Password

[ ]:
```python
@app.route('/provider-reset-password/<token>', methods=['GET', 'POST'])
def provider_reset_password(token):
    conn, cursor = get_db_connection()
    if not conn or not cursor:
        flash('Database connection error', 'error')
        return redirect(url_for('provider_login'))

    try:
        # Join provider_password_reset_tokens with service_providers table to␣
↪get provider information
        cursor.execute('''
            SELECT sp.id, sp.email
            FROM provider_password_reset_tokens t
            JOIN service_providers sp ON t.provider_id = sp.id
            WHERE t.token = %s AND t.expiry > NOW()
        ''', (token,))
        result = cursor.fetchone()

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

                if new_password == confirm_password:
                    # Update provider's password
                    cursor.execute('UPDATE service_providers SET password = %s␣
↪WHERE id = %s',
                                   (new_password, result['id']))
                    # Delete the used token
                    cursor.execute('DELETE FROM provider_password_reset_tokens␣
↪WHERE token = %s', (token,))
                    conn.commit()
```

```
                    flash('Your password has been updated successfully.',␣
↪'success')
                    return redirect(url_for('provider_login'))
                else:
                    flash('Passwords do not match. Please try again.', 'error')

            return render_template('provider_reset_password.html', token=token)
        else:
            flash('Invalid or expired reset token. Please try again.', 'error')
            return redirect(url_for('provider_forgot_password'))
    except mysql.connector.Error as err:
        flash(f'An error occurred: {err}', 'error')
    finally:
        cursor.close()
        conn.close()

    return redirect(url_for('provider_login'))
```

Route: /provider-reset-password/

Methods: GET, POST

Description: Allows providers to reset their password by validating a secure token sent via email.

Functionality:

1. Validates the token and checks its expiry.

2. Fetches the associated provider based on the token.

3. Accepts new password and confirmation from the provider.

4. Updates the provider's password if the token is valid and the passwords match.

5. Deletes the used token from the database.

Returns:

1. On success: Redirects to provider login page with a success message.

2. On failure: Displays error messages and redirects to the forgot password page if the token

1.8 Women Service Provider-Related Routing

1. Women service provider registration

[ ]:
```
@app.route('/women-provider-register', methods=['GET', 'POST'])
def women_provider_register():
    if request.method == 'POST':
        name = request.form.get('name')
        username = request.form.get('username')
```

```
        email = request.form.get('email')
        domain = request.form.get('domain')
        phone = request.form.get('phone')
        password = request.form.get('password')
        confirm_password = request.form.get('confirm_password')
        age = request.form.get('age')
        gov_id = request.form.get('gov_id')

        conn, cursor = get_db_connection()
        if not conn or not cursor:
            return "Database connection error", 500

        try:
            # Insert new provider into the women_service_providers table
            cursor.execute(
                "INSERT INTO women_service_providers (name, username, email,␣
↪domain, phone, password, age, gov_id) VALUES (%s, %s, %s, %s, %s, %s, %s,␣
↪%s)",
                (name, username, email, domain, phone, password, age, gov_id)
            )
            conn.commit()
            return redirect(url_for('women_login'))
        except mysql.connector.Error as err:
            return render_template('women_provider_registration.html', error =␣
↪"Username or email already exists.")

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

Route: /women-provider-register

Methods: 1. GET: Renders the registration form for a new service provider. 2. POST: Processes the registration form and adds the new provider to the database.

Code Explanation: 1. GET Request:

`When the route is accessed via GET, the women_provider_registration.html form is rendered for r`

2. POST Request:

   When the form is submitted, the route collects data from the form fields:

   `name, username, email, domain, phone, password, confirm_password, age, gov_id.`

   A connection to the database is established via get_db_connection().

   If the connection fails, a “Database connection error” is returned with a 500 status code.

   If successful, it attempts to insert the new provider’s details into the women_service_providers table.

   If the insertion is successful, the user is redirected to the login page (women_login).

If there's a MySQL error (e.g., the username or email already exists), the registration form is re-rendered with an error message: "Username or email already exists."

Returns:

1. On GET: Renders the registration page women_provider_registration.html.
2. On POST: Redirects to the login page (women_login) on successful registration or re-renders the form with an error if the registration fails.

2. Women service provider Home page

```
[ ]: @app.route('/women-home')
def women_home():
    # Check if the user is logged in and has the correct role
    if 'username' not in session or session.get('user_role') !=␣
 ↪'women_provider':
        return redirect(url_for('women_login'))  # Redirect to login if not␣
 ↪logged in or incorrect user role

    # Pass the username to the template
    return render_template('women_home.html', username=session['username'])
```

Route: /women-home

Methods:

1. GET: Displays the home page for a logged-in women service provider.

Code Explanation:

1. User Authentication and Role Check:
    1. The function checks whether the user is logged in by verifying the presence of the username key in the session.
    2. It also ensures that the logged-in user has the correct role (women_provider), which is stored in the user_role session key.
    3. If the user is not logged in or does not have the correct role, they are redirected to the login page (women_login).
2. Rendering the Home Page:
    1. If the user passes the authentication check, the function renders the women_home.html template.
    2. The logged-in provider's username, stored in the session, is passed to the template to display personalized information on the home page.

Returns:

1. If the user is authenticated and authorized: Renders the women_home.html template with the username passed as a context variable.
2. If the user is not authenticated or has the wrong role: Redirects the user to the women_login page for authentication.

3. Women service provider login page

```
[ ]: @app.route('/women-login', methods=['GET', 'POST'])
def women_login():
    error = None
    if request.method == 'POST':
        username = request.form.get('username')
        password = request.form.get('password')

        # Get DB connection
        conn, cursor = get_db_connection()
        if not conn or not cursor:
            return "Database connection error", 500

        # Use dictionary cursor
        cursor = conn.cursor(dictionary=True)

        # Query to check user credentials
        cursor.execute(
            "SELECT * FROM women_service_providers WHERE username = %s AND␣
 ↪password = %s",
            (username, password)
        )
        provider = cursor.fetchone()

        # If provider exists, set session variables
        if provider:
            session['provider_id'] = provider['id']
            session['username'] = provider['username']
            session['user_role'] = 'women_provider'  # Set user role for women␣
 ↪service provider
            return redirect(url_for('women_home'))
        else:
            error = "Invalid username or password"

    return render_template('women_login.html', error=error)
```

Route: /women-login

Methods: 1. GET: Displays the login page for women service providers. 2. POST: Processes login information submitted via a form.

Code Explanation: 1. Form Submission (POST):

When the form is submitted, the function retrieves the username and password from the form usi

2. Database Connection:

    1. The function establishes a connection to the database using get_db_connection().
    2. If the connection fails, a 500 error is returned indicating a database connection issue.

3. Query to Validate User:
    1. The function uses a dictionary cursor (conn.cursor(dictionary=True)) to fetch user data in the form of a dictionary.
    2. It runs an SQL query to check if the provided username and password match an entry in the women_service_providers table.
    3. If a match is found, the user's details are retrieved.
4. Session Management:
    1. If the provider exists: The provider_id, username, and user_role (set to women_provider) are stored in the session for future access control. The user is redirected to the /women-home route, which serves the provider's dashboard.
    2. If the credentials are invalid, an error message is set: "Invalid username or password", which is passed to the template for display.
5. Rendering Login Page:
    1. The function renders the women_login.html template.
    2. If there is an error (like invalid credentials), the error message is passed to the template to inform the user.

Returns: 1. If the user credentials are valid: Redirects the user to the /women-home route. 2. If the user credentials are invalid or no credentials provided (GET request): Renders the women_login.html template, optionally with an error message.

4. Women service provider profile page

[ ]:
```
@app.route('/women-provider-profile', methods=['GET', 'POST'])
def women_provider_profile():
    if 'username' not in session:
        flash('Please log in to access your profile.', 'error')
        return redirect(url_for('women_provider_login'))

    conn, cursor = get_db_connection()
    if not conn or not cursor:
        flash('Database connection error. Please try again later.', 'error')
        return render_template('women_provider_profile.html')

    provider = None
    try:
        cursor.execute('SELECT * FROM women_service_providers WHERE username =␣
↪%s', (session['username'],))
        provider = cursor.fetchone()

        if request.method == 'POST':
            name = request.form.get('name')
            phone = request.form.get('phone')
            address = request.form.get('address')
```

```
            if provider:
                cursor.execute('UPDATE women_service_providers SET name = %s,␣
↪phone = %s, address = %s WHERE username = %s',
                               (name, phone, address, session['username']))
            else:
                cursor.execute('INSERT INTO women_service_providers (username,␣
↪name, phone, address) VALUES (%s, %s, %s, %s)',
                               (session['username'], name, phone, address))
            conn.commit()
            flash('Profile updated successfully!', 'success')

            cursor.execute('SELECT * FROM women_service_providers WHERE␣
↪username = %s', (session['username'],))
            provider = cursor.fetchone()

    except mysql.connector.Error as err:
        conn.rollback()
        flash(f'An error occurred: {err}', 'error')
    finally:
        cursor.close()
        conn.close()

    return render_template('women_provider_profile.html', provider=provider)
```

Route: /women-provider-profile

Methods: 1. GET: Displays the profile page for the logged-in women service provider. 2. POST: Updates the profile information based on form input.

Code Explanation: 1. Session Check:

```
1. The function first checks if the username is present in the session, ensuring the user is lo
2. If not, it flashes an error message, prompting the user to log in, and redirects to the wome
```

2. Database Connection:
    1. The function establishes a database connection using get_db_connection().
    2. If the connection fails, an error message is flashed and the profile page is rendered with no data (None for provider).
3. Fetching Provider Information (GET Request):

    For a GET request, the function fetches the current profile data of the logged-in provider from the women_service_providers table using the username stored in the session.
4. Updating Provider Information (POST Request):
    1. When the user submits a form via POST, the function retrieves the updated name, phone, and address from the form.
    2. If the provider's profile exists:

An SQL UPDATE query is executed to update the provider's information based on the username stored in the session.

3. If no profile exists for the provider:

   A new profile entry is inserted using an INSERT INTO query with the provided details.

4. After successfully updating or inserting the profile, the transaction is committed and a success message is flashed.

5. Error Handling:

   If an error occurs during the SQL operations, the transaction is rolled back to prevent partial updates, and an error message is flashed with details.

6. Re-fetching Updated Data:

   After a successful update, the provider's updated profile information is fetched again from the database and passed to the template for display.

7. Rendering the Template:

   The women_provider_profile.html template is rendered with the provider's profile information (provider), allowing the user to see their updated details.

Flash Messages: 1. Error: Displayed when there's an issue with the database connection or an SQL error during profile update. 2. Success: Displayed when the profile is updated successfully.

Returns: 1. Renders the profile page (women_provider_profile.html) with the provider's current or updated data.

5. Women service provider update password functionality

[ ]:
```
@app.route('/women-provider-update-password', methods=['GET', 'POST'])
def women_provider_update_password():
    if 'username' not in session:
        flash('You must be logged in to change your password.', 'error')
        return redirect(url_for('women_provider_login'))

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

        if not all([current_password, new_password, confirm_password]):
            flash('All fields are required.', 'error')
            return redirect(url_for('women_provider_update_password'))

        if new_password != confirm_password:
            flash('New passwords do not match. Please try again.', 'error')
            return redirect(url_for('women_provider_update_password'))

        conn, cursor = get_db_connection()
```

```
        if not conn or not cursor:
            flash('Database connection error', 'error')
            return redirect(url_for('women_provider_update_password'))

        try:
            cursor.execute('SELECT * FROM women_service_providers WHERE␣
↪username = %s', (session['username'],))
            provider = cursor.fetchone()

            if provider:
                stored_password = provider['password']
                if current_password == stored_password:
                    cursor.execute('UPDATE women_service_providers SET password␣
↪= %s WHERE username = %s',
                                   (new_password, session['username']))
                    conn.commit()
                    flash('Password updated successfully!', 'success')
                else:
                    flash('Current password is incorrect. Please try again.',␣
↪'error')
            else:
                flash('Provider not found.', 'error')

        except mysql.connector.Error as err:
            conn.rollback()
            flash(f'An error occurred: {err}', 'error')
        finally:
            if cursor:
                cursor.close()
            if conn:
                conn.close()
    return render_template('women_provider_profile.html', provider = provider)
```

Route: /women-provider-update-password

Functionality: 1. GET: Displays the password update form. 2. POST: Processes the password change request.

Key Steps: 1. Session Check:

`If the user isn't logged in, they are redirected to the login page with an error message.`

2. Form Submission (POST):

   Ensures all form fields (current_password, new_password, confirm_password) are filled.

   Validates that new_password matches confirm_password.

3. Database Interaction:

   Fetches the current provider's data using the username from the session.

Compares the current_password with the stored password.

If correct, updates the password in the database.

4. Flash Messages:

   Success or error messages are displayed based on validation and database operation outcomes.

5. Rendering:

   Renders the women_provider_profile.html template with relevant messages.

Flash Messages: 1. Errors: For missing fields, password mismatch, incorrect current password, or database issues. 2. Success: For successful password updates.

6. Bookings for women service provider

[ ]:
```
@app.route('/women-provider-bookings')
def women_provider_bookings():
    if 'username' not in session:
        return redirect(url_for('women_provider_login'))
    return render_template('women_provider_bookings.html')

@app.route('/api/women-provider-bookings')
def api_women_provider_bookings():
    logging.info("Entering api_women_provider_bookings function")
    conn, cursor = None, None
    try:
        # Log the entire session data to ensure the username is set
        logging.info(f"Session data: {session.items()}")

        conn, cursor = get_db_connection()
        if not conn or not cursor:
            logging.error("Failed to establish a database connection.")
            return jsonify({"error": "Database connection error"}), 500

        username = session.get('username')
        logging.info(f"Provider username from session: {username}")
        if not username:
            logging.warning("User is not logged in.")
            return jsonify({"error": "User not logged in"}), 401

        # Check if the provider exists in the service_providers table
        check_provider_query = "SELECT * FROM women_service_providers WHERE␣
 ↪username = %s"
        cursor.execute(check_provider_query, (username,))
        provider = cursor.fetchone()

        if not provider:
            logging.warning(f"No provider found with username: {username}")
            return jsonify({"error": "Provider not found"}), 404
```

```
        logging.info(f"Provider found: {provider}")

        # Fetch the bookings for the provider
        query = """
            SELECT * FROM women_service_appointments
            WHERE provider = %s
            ORDER BY date DESC, time DESC
        """
        cursor.execute(query, (provider['name'],))
        rows = cursor.fetchall()

        logging.info(f"Query executed. Number of rows returned: {len(rows)}")

        if rows:
            bookings = []
            for row in rows:
                # Log the raw row data to see exactly what is being fetched
                logging.info(f"Raw row data: {row}")

                # Ensure that the client's username is correctly extracted
                client_username = row.get('username', 'Unknown Client')
                if client_username == 'Unknown Client':
                    logging.warning(f"Client username missing for booking ID:␣
↪{row['id']}")

                # Build the booking dictionary
                booking = {
                    "id": row['id'],
                    "date": row['date'].strftime('%Y-%m-%d'),
                    "time": str(row['time']),
                    "service": row['service'],
                    "provider": row['provider'],
                    "notes": row.get('notes', ''),
                    "client_username": client_username
                }
                bookings.append(booking)
                logging.info(f"Processed booking: {json.dumps(booking)}")

            logging.info(f"Bookings fetched for provider: {json.dumps(bookings,␣
↪indent=2)}")
            return jsonify({"bookings": bookings})
        else:
            logging.info(f"No bookings found for the provider: {username}")
            return jsonify({"message": "No bookings found"}), 200
    except Exception as e:
```

```
        logging.exception(f"An unexpected error occurred in women provider␣
↪bookings: {str(e)}")
        return jsonify({"error": str(e)}), 500
    finally:
        if cursor:
            cursor.close()
        if conn:
            conn.close()
        logging.info("Exiting api_women_provider_bookings function")
```

1. Route: /women-provider-bookings (GET)

   Purpose: Serves the bookings page for women service providers.

   Session Check: Verifies if the user is logged in, otherwise redirects to the login page.

   Template Rendering: Renders the women_provider_bookings.html page.

2. Route: /api/women-provider-bookings (GET)

   Purpose: Provides an API endpoint to fetch a provider's bookings.

   Session Check: Ensures the user is logged in by checking the session for username.

   Database Interaction:

   ```
   1. Fetches the provider's data using the username from the session.

   2. Retrieves the provider's bookings from the women_service_appointments table.

   3. Each booking includes details like date, time, service, provider, notes, and client_us
   ```

   Logging:

   ```
   Extensive logging is used to trace the session data, query execution, and any potential e

   Logs details such as session contents, provider data, fetched rows, and individual bookir
   ```

   Error Handling:

   ```
   Returns appropriate error responses if the user is not logged in, the provider is not fou
   ```

   JSON Response: Sends a JSON response with a list of bookings or a message indicating no bookings were found.

This API and webpage are designed to help the women providers view and manage their service bookings efficiently.

7. Women service provider earnings

[ ]:
```
@app.route('/women-provider-earnings', methods=['GET', 'POST'])
def women_provider_earnings():
    conn, cursor = get_db_connection()
    username = session.get('username')  # Get provider username from session
```

```
    if not username:
        print("Provider name is not set in session.")
        return "Error: Provider name is not set in session."

    # Handle POST request to calculate and add earnings
    if request.method == 'POST':
        try:
            hourly_charge = float(request.form['hourlyCharge'])  # Get hourly␣
↪charge
            hours_worked = float(request.form['hoursWorked'])  # Get hours␣
↪worked
            session_earnings = Decimal(hourly_charge * hours_worked)  #␣
↪Calculate session earnings

            # Check if provider already has an entry
            query = "SELECT earnings FROM women_provider_earnings WHERE␣
↪provider_name = %s"
            cursor.execute(query, (username,))
            result = cursor.fetchone()

            if result:
                # Convert existing earnings to Decimal and update
                existing_earnings = Decimal(result['earnings'])
                total_earnings = existing_earnings + session_earnings
                update_query = "UPDATE women_provider_earnings SET earnings =␣
↪%s WHERE provider_name = %s"
                cursor.execute(update_query, (total_earnings, username))
            else:
                # Insert new entry for the provider
                total_earnings = session_earnings
                insert_query = "INSERT INTO women_provider_earnings␣
↪(provider_name, earnings) VALUES (%s, %s)"
                cursor.execute(insert_query, (username, total_earnings))

            conn.commit()

            # Redirect to display updated earnings
            return redirect(url_for('women_provider_earnings'))

        except Exception as e:
            print(f"An error occurred: {e}")
            return "Error processing your request.", 500

    # Handle GET request to display the earnings page
    query = "SELECT earnings FROM women_provider_earnings WHERE provider_name =␣
↪%s"
```

```
    cursor.execute(query, (username,))
    result = cursor.fetchone()

    if result and 'earnings' in result:
        total_earnings = Decimal(result['earnings'])
    else:
        total_earnings = Decimal('0.00')  # Default if no record is found

    # Render the earnings page with total earnings
    return render_template('women_provider_earnings.html',␣
↪total_earnings=total_earnings)
```

Route: /women-provider-earnings (GET, POST)

Purpose: Manages and displays the earnings for a women service provider.

Session Check:

1. Retrieves the provider's username from the session to associate earnings with the correct pr

2. If no username is found, it returns an error.

GET Request Handling:

1. Queries the women_provider_earnings table for the provider's earnings.

2. If a record is found, it extracts the earnings value; otherwise, it defaults to 0.00.

3. Renders the women_provider_earnings.html page, displaying the provider's total earnings.

POST Request Handling:

1. Receives form data (hourlyCharge and hoursWorked).

2. Calculates session earnings by multiplying hourly rate and hours worked.

3. Database Interaction:

    Checks if an entry for the provider already exists in the women_provider_earnings table:

        If the provider exists, it updates the earnings by adding the new session earnings to t

        If no record exists, it inserts a new entry with the calculated earnings.

4. Commits changes to the database.

5. Redirects to the same page to display updated earnings.

Error Handling:

Catches exceptions during the process and logs any errors to help troubleshoot.

In case of an error, it returns a message with an HTTP 500 status.

This route helps the provider track their earnings by allowing them to log session earnings and view cumulative earnings.

8. Women service provider forgot and reset password functionality

```
[ ]: @app.route('/women-forgot-password', methods=['GET', 'POST'])
def women_forgot_password():
    if request.method == 'POST':
        email = request.form['email']
        conn, cursor = get_db_connection()
        if not conn or not cursor:
            flash('Database connection error', 'error')
            return redirect(url_for('women_forgot_password'))

        try:
            cursor.execute('SELECT id FROM women_service_providers WHERE email␣
 ↪= %s', (email,))
            provider = cursor.fetchone()

            if provider:
                token = secrets.token_urlsafe(32)
                expiry = datetime.now() + timedelta(hours=1)

                cursor.execute('INSERT INTO women_password_reset_tokens␣
 ↪(provider_id, token, expiry) VALUES (%s, %s, %s)',
                               (provider['id'], token, expiry))
                conn.commit()

                reset_link = url_for('women_reset_password', token=token,␣
 ↪_external=True)
                send_reset_email(email, reset_link)

                flash('Password reset link has been sent to your email.',␣
 ↪'success')
                return redirect(url_for('women_login'))
            else:
                flash('Email not found. Please check and try again.', 'error')
        except mysql.connector.Error as err:
            flash(f'An error occurred: {err}', 'error')
        finally:
            cursor.close()
            conn.close()

    return render_template('women_forgot_password.html')

def send_reset_email(email, reset_link):
```

```
    msg = Message('Password Reset Request',
                  sender='nirmaldummy@gmail.com',
                  recipients=[email])
    msg.body = f'''To reset your password, visit the following link:␣
 ↪{reset_link}

If you did not make this request then simply ignore this email and no changes␣
 ↪will be made.
'''
    mail.send(msg)

@app.route('/women-reset-password/<token>', methods=['GET', 'POST'])
def women_reset_password(token):
    conn, cursor = get_db_connection()
    if not conn or not cursor:
        flash('Database connection error', 'error')
        return redirect(url_for('women_login'))

    try:
        # Join women_password_reset_tokens with women_service_providers table␣
 ↪to get provider information
        cursor.execute('''
            SELECT wsp.id, wsp.email
            FROM women_password_reset_tokens t
            JOIN women_service_providers wsp ON t.provider_id = wsp.id
            WHERE t.token = %s AND t.expiry > NOW()
        ''', (token,))
        result = cursor.fetchone()

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

                if new_password == confirm_password:
                    # Update provider's password
                    cursor.execute('UPDATE women_service_providers SET password␣
 ↪= %s WHERE id = %s',
                                   (new_password, result['id']))
                    # Delete the used token
                    cursor.execute('DELETE FROM women_password_reset_tokens␣
 ↪WHERE token = %s', (token,))
                    conn.commit()
                    flash('Your password has been updated successfully.',␣
 ↪'success')
                    return redirect(url_for('women_login'))
```

```
                else:
                    flash('Passwords do not match. Please try again.', 'error')

            return render_template('women_reset_password.html', token=token)
        else:
            flash('Invalid or expired reset token. Please try again.', 'error')
            return redirect(url_for('women_forgot_password'))
    except mysql.connector.Error as err:
        flash(f'An error occurred: {err}', 'error')
    finally:
        cursor.close()
        conn.close()

    return redirect(url_for('women_login'))
```

1. Route: /women-forgot-password (GET, POST)

Purpose: Allows women service providers to request a password reset by entering their registered email address.

POST Request Handling:

Retrieves the submitted email.

Checks if the email exists in the women_service_providers table.

If found:

    Generates a secure token using secrets.token_urlsafe(32) and sets an expiry time of one hou

    Inserts the token and its expiry time into the women_password_reset_tokens table.

    Sends an email to the user with the password reset link.

If the email is not found, a flash message alerts the user.

GET Request Handling:

Displays the password reset request form (women_forgot_password.html).

Helper Function:

send_reset_email: Sends a password reset email to the user containing the reset link.

Uses Flask-Mail to send the email.

2. Route: /women-reset-password/ (GET, POST)

Purpose: Handles the actual password reset process after the user clicks the link in their email.

Token Validation:

Validates the token by checking the women_password_reset_tokens table to ensure it exists and

If valid, it joins the token table with the women_service_providers table to fetch the provide

POST Request Handling:

After the token is validated, the user can input a new password.

If the passwords match, the provider's password is updated in the women_service_providers table

The used token is then deleted from women_password_reset_tokens to prevent reuse.

GET Request Handling:

Displays the reset password form (women_reset_password.html) for the user to input a new passwo

Token Expiry or Invalid Token:

If the token is invalid or expired, a message is flashed, and the user is redirected to request

Key Functionalities

1. Token Generation: The token is securely generated and stored with an expiry to prevent unauthorized access.
2. Email Notification: A reset link is sent to the registered email address.
3. Password Update: Once the new password is confirmed, it is securely updated in the database.
4. Token Expiry Handling: Ensures tokens can only be used within their valid time frame to enhance security.

1.9 Flask Application Initialization Command

[ ]:
```python
### App running command
if __name__ == '__main__':
    app.run(debug=True)
```

This is the standard command to run a Flask application. Here's what it does:

Explanation:

1. if **name** == '**main**':

   This ensures that the Flask app only runs when the script is executed directly (i.e., not when imported as a module).

2. app.run(debug=True)

   app.run(): Starts the Flask development server.

   debug=True: Enables debug mode, which provides useful error messages, auto-reloads the server on code changes, and generally makes development smoother.

This command starts your Flask app in development mode, making it easy to identify and troubleshoot errors during development.

1.10 Conclusion

In this project, "Grihaseva," I have developed a robust Flask application that efficiently manages user appointments and provides seamless navigation for both users and service providers. Through careful design and implementation, I have ensured that the application not only meets functional requirements but also delivers a user-friendly experience. By utilizing technologies such as MySQL for data management, Axios and Fetch for API interactions, and a responsive front-end, Grihaseva stands ready to enhance the appointment booking process. I am excited about the potential for further enhancements and the positive impact this application will have on the users.