Object-Oriented Programming
Baptiste KELAGOPIAN

BIRTHDAY REMINDER

Introduction

The goal of this project is to create a web application that helps users remember their friends' birthdays by sending email reminders. The application, called "Birthday Reminder," allows users to register and log in to access two main features: adding friends' birthdays and viewing a list of upcoming birthdays. To add a friend's birthday, the user simply needs to enter their name and birthday, and the application will store the information in a MySQL database. The user can also choose to receive an email reminder on the day of their friend's birthday. The application will display a list of all upcoming birthdays, and the user can choose to receive email reminders for any of them. The application is built using Flask, a Python web framework, and utilizes the Flask-Mail extension to send email reminders to users. The user can then register for an account, log in, and start adding birthdays to the database. On the day of a friend's birthday, if the user has chosen to receive an email reminder, the application will send an email to remind them to wish their friend a happy birthday.

How to run the program :
The user needs to have Python and Flask installed on their computer and set up a MySQL database.
Here is a SQL script to create the Tables at it should be :

```
CREATE TABLE IF NOT EXISTS birthdays (
    id INT(11) NOT NULL AUTO_INCREMENT,
    account_id INT(11) NOT NULL,
    name VARCHAR(255) NOT NULL,
    date DATE NOT NULL,
    PRIMARY KEY (id),
    FOREIGN KEY (account_id) REFERENCES accounts(id) ON DELETE CASCADE ON
UPDATE CASCADE
) ENGINE=InnoDB AUTO_INCREMENT=1 DEFAULT CHARSET=utf8;
```

And

```
CREATE TABLE IF NOT EXISTS accounts (
    id INT(11) NOT NULL AUTO_INCREMENT,
    password VARCHAR(255) NOT NULL,
    email VARCHAR(255) NOT NULL,
    PRIMARY KEY (id)
) ENGINE=InnoDB AUTO_INCREMENT=2 DEFAULT CHARSET=utf8;
```

To run :

```
$flask --app app --debug run
```

You will find all of the configurations in **config.py**

The program covers the following functional requirements:

Inheritance:

The class BirthdayAddReminderEmail is inheriting from the class BirthdayReminderEmail. This is evident by the notation class BirthdayAddReminderEmail(BirthdayReminderEmail). Inheritance allows one class to gain all the members (say attributes and methods) of another. In this case, BirthdayAddReminderEmail gains all the attributes and methods of BirthdayReminderEmail.

Polymorphism:

The send method is defined in both BirthdayReminderEmail and BirthdayAddReminderEmail. This is an example of method overriding, a form of polymorphism. Polymorphism allows methods to behave differently, based on the object that it is acting upon. In this case, send behaves differently for objects of BirthdayReminderEmail and BirthdayAddReminderEmail.

```
class BirthdayReminderEmail:
    def __init__(self, recipient, app):
        self.app = app
        self.sender = "Birthday Reminder"
        self.recipient = recipient
        self.__executors = {'default': ThreadPoolExecutor(20)}
        self.scheduler = BackgroundScheduler(executors=self.__executors)
        self.scheduler.start()

    def send(self, name):
        with self.app.app_context():
            msg = Message('Birthday Reminder', sender=self.sender,
recipients=[self.recipient])
            msg.html = f"<html><body>Hello,<br><br>Today is the birthday of
<span style='color: red;'>{name}</span><br>Wish him/her a happy
birthday!<br><br>Have a nice day!</body></html>"
```

```
            mail.send(msg)

class BirthdayAddReminderEmail(BirthdayReminderEmail):
    def send(self, name, date):
        with self.app.app_context():
            msg = Message(f'Birthday Reminder, you added {name}',
sender=self.sender, recipients=[self.recipient])
            msg.html = f"<html><body>Hello,<br><br>You just added to your
reminder the birthday of <span style='color: red;'>{name}</span> on <span
style='color: blue;'>{date}</span><br>We will send you a reminder on the day of
the event.<br><br>Have a nice day!</body></html>"
            mail.send(msg)
            self.scheduler.add_job(self.send_reminder, 'date', run_date=date,
args=[self.recipient, name])

    def send_reminder(self, email, name):
        super().send(name)
```

Abstraction:

```
from abc import ABC, abstractmethod

class DatabaseModel(ABC):
    def __init__(self, cursor):
        self.cursor = cursor

    @abstractmethod
    def insert(self):
        pass

    @abstractmethod
    def get(self):
        pass
```

The DatabaseModel class is an abstract class that provides a generic interface for interacting with a database. The insert and get methods are abstract methods that must be implemented by concrete subclasses.

Encapsulation:

```
class Birthday(DatabaseModel):
    def __init__(self, name, date, account_id, cursor):
        super().__init__(cursor)
        self.name = name
        self.date = date
        self.account_id = account_id

class Account(DatabaseModel):
    def __init__(self, email, password, cursor):
        super().__init__(cursor)
        self.email = email
        self.password = password
```

The DatabaseModel, Birthday and Account classes encapsulate their own state. For example, Birthday encapsulates name, date and account_id, while Account encapsulates email and password. These states can only be accessed via the methods of these classes.

Design patterns

Decorator :

login_required est une fonction qui prend une autre fonction func en paramètre et retourne une fonction interne wrapper qui enveloppe func. La fonction wrapper vérifie si l'utilisateur est connecté en utilisant session.get('loggedin'). Si l'utilisateur n'est pas connecté, la fonction wrapper redirige l'utilisateur vers la page de connexion en utilisant redirect(url_for("login")).

```
def login_required(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        if session.get('loggedin') is not True:
            return redirect(url_for("login"))
        return func(*args, **kwargs)
    return wrapper

@app.route("/remind_me", methods = ['GET', 'POST'])
```

```
@login_required
def remind_me():
    if request.method == 'POST':
        name = request.form['name']
        date = request.form['date']
        emailAddReminder = BirthdayAddReminderEmail(session['email'], app)
        emailAddReminder.send(name, date)
        flash('Email sent successfully')
        return redirect(url_for('index'))
    return redirect(url_for('index'))
```

Singleton

In the code provided, there's an example of a Singleton. The DatabaseService class is implemented as a Singleton using the __new__() approach. This approach guarantees that there will be only one instance of the DatabaseService class in the application.

```
class DatabaseService:
    _instance = None

    def __new__(cls, *args, **kwargs):
        if not cls._instance:
            cls._instance = super(DatabaseService, cls).__new__(cls)
        return cls._instance
```

Reading from file & writing to file

User registration and login: The user can register an account and log in using their email and password. The account information is stored in the MySQL database.

The register() function handles user registration by inserting the user's email and password into the MySQL database.

```
class Account(DatabaseModel):
   . . .
    def insert(self):
        self.cursor.execute('INSERT INTO accounts (email, password) VALUES (%s,
%s)', (self.email, self.password,))
        mysql.connection.commit()
```

The login() function handles user login by checking if the provided email and password match an account in the database.

```
class Account(DatabaseModel):
   . . .
    def get(email, password):
        cursor = mysql.connection.cursor(cursorclass=DictCursor)
        cursor.execute('SELECT * FROM accounts WHERE email = %s AND password =
%s', (email, password,))
        return cursor.fetchone()
```

Testing

Unit tests are implemented using the pytest library to verify the behavior of the Flask application in the unitTesting.py file.

Example :

The test_login() function tests the application's login functionality. It sends a POST request to the "/login" endpoint with valid e-mail and password data. The expected response is an HTTP 200 status code and the presence of the string "Be a better friend and stop forgetting they birthdays" in the body of the response, which is present in the main page.

```
def test_login(client):
    email = "test@gmail.com"
    password = "test"

    response = client.post("/login", data={
        "email": email,
        "password": password
    })

    assert response.status_code == 200
    assert b"Be a better friend and stop forgetting they birthdays" in
response.data
```

The test_wrong_login() function tests the application's login functionality with invalid e-mail and password data. It sends a POST request to the "/login" endpoint with incorrect e-mail and password data. The expected response is that the string "Be a better friend and stop forgetting they birthdays" is not present in the body of the response.

```
def test_wrong_login(client):
    email = "wrong@example.com"
    password = "wrongpassword"

    response = client.post("/login", data={
        "email": email,
        "password": password
```

```
    })
    assert b"Be a better friend and stop forgetting they birthdays" not in
response.data
```

How to run

```
$python -m unittest .\unitTesting.py
```

Results

The program successfully implements the functional requirements of user registration, login, adding a friend's birthday, and sending an email reminder.

- The implementation of the Birthday Reminder application was challenging as it was the developer's first time creating a web application in Python. However, the developer was able to successfully implement all the required features of the application.
- One of the main challenges faced during implementation was ensuring the user-friendliness of the application. The developer focused on creating an intuitive and easy-to-use interface for the user, with clear instructions and minimal clutter.
- To achieve this, the developer used a clean and simple design for the application, with clear labels and buttons for each action. The user is able to easily navigate between the different pages of the application, and the process of adding a new friend and their birthday is straightforward and self-explanatory.

Overall, the developer was successful in creating a user-friendly Birthday Reminder application that is easy to navigate and use, even for those who may not be familiar with web applications.

Conclusions

The program provides a simple and useful solution for remembering friends' birthdays. It allows the user to register, log in, add birthdays, and receive email reminders. The program can be extended in several ways, such as adding the ability to edit or delete birthdays, or integrating with a calendar application. Overall, the program demonstrates the use of Flask, MySQL, and Flask-Mail to create a web application with database and email functionality.