Profile-Management

This Document is meant to Hardway of HTTP Methods to manage the database of user records, allowing for the addition, modification, partial updates, and deletion of records and reseting the password.

PROJECT STRUCTURE

```
profile-management/
├── Design Docs/
|       └── Profile-Management
├── apps/
|       ├── __init__.py
|       ├── app.py
|       ├── constants.py
├── emails/
|       ├── __init__.py
|       ├── email_operations.py
├── log/
|       ├── __init__.py
|       ├── app.log
|       ├── logging_logic.py
├── utilis/
|       ├── __init__.py
|       ├── utility.py
```

APPS/CONSTANTS.PY

This contains the constant variables and email configuration messages that are used in the application.

```python
"""
This module defines constants used throughout the application
"""
DATA = {"records": [
    {'username': 'kumar123', 'name': 'kumar.M', 'dept': 'DEV', 'dob': '2001-2-2', 'ger
     'password': 'Kumar@123', 'isadmin': False},
    {'username': 'dinesh2003', 'name': 'dinesh.A', 'dept': 'APP', 'dob': '2001-2-3', '
     'password': 'Dinesh&2003', 'isadmin': False}
]}
VALID_GENDER = ["MALE", "FEMALE", "OTHERS", "M", "F"]
USERS = ["kumar.M", "dinesh.A"]
ADMINS = ["Donlee.L"]
ALL_USERS = ["kumar.M", "dinesh.A", "Donlee.L"]

ADMIN_CONSOLE = f"Which Operation you need to perform \n1)Create User \n2)Update Recor

USER_CONSOLE = f'Which operation need to perform \n1)Create User \n2)Update your recor

# emails configurations
SMTP_PORT = 587
SMTP_SERVER = "smtp.gmail.com"
SENDER_EMAIL = "dineshsai14211@gmail.com"
RECEIVER_EMAIL = ["dineshsai14211@gmail.com", "danepaku@gitam.in"]
```

```
PASSWORD = "izdlrkcuyzrlwxec"

DELETE_MESSAGE = """
Hlo Team,\n
Admin={} has deleted user {} Record in DATA
Thank you
"""
UPDATE_MESSAGE = """
Hlo Team,\n
Admin={} has updated user={} record.
Record = {}\n
Thank You
"""
GET_ALL_MESSAGE = """
Hlo Team,\n
User={} , is checking all users information from the table.\n
Thank You
"""
PARTIAL_UPDATE = """
Hlo Team\n
Admin={} has partially updated the user={} records in DATA.\n
Partially updated user record = {}\n
Thank you
"""
UNAUTHENTICATED_MESSAGE = """
Hlo team\n
ALERT:-Unauthenticated user={}, trying to access the application
check logs\n
Thank You
"""
UPDATE_USER_RECORD = """
Hlo team\n
''User={} has updated his profile''
Updated Record = {}
\nThank you
"""
RECORD_ADDED = """
Hlo Team,\n
Admin={} added the user={} record = {}.\n
Thank You
"""
RESET_PASSWORD = """
Hlo Team\n
Admin={} has reseted the user={} password\n
Thank You
"""
CHECK_USER_INFO = """
Hlo Team,\n
Admin={} checking user={} information.\n
Thank You
"""
```

LOG/LOGGING_LOGIC.PY

Importing the logging module as '**log**' and configuring the log message format, logs will be recorded in the '**app.log**' file.

```
import logging as log

log.basicConfig(filename="../log/app.log", filemode="a", level=log.DEBUG,
                format="%(asctime)s - %(levelname)s - %(message)s")
```

EMAILS/EMAIL_OPERATIONS.PY

- This code imports the '**smtplib**' and '**ssl**' modules, along with constants from the '**apps.constants**' module.
- **send_email** function checks whether the too_email parameter is provided or not. If not, it defaults to an empty list. The function then establishes a secure connection to the SMTP server ,login with the sender's email and password, and sends the email_body to the specified recipients.

```
import smtplib
import ssl

from apps.constants import *

sender = SENDER_EMAIL
receivers = RECEIVER_EMAIL
context = ssl.create_default_context()


def send_email(too_email=None, email_body=""):
    if too_email is None:
        too_email = []
    with smtplib.SMTP(SMTP_SERVER, SMTP_PORT) as server:
        server.starttls(context=context)
        server.login(sender, PASSWORD)
        server.sendmail(sender, too_email, email_body)
```

UTILIS/UTILITY.PY

Importing all constants variable and regular expression, datatime module, logging functionality.

```
from datetime import datetime
import re
from apps.constants import *
from log.logging_logic import *
```

is_valid_name function is used to validate the name. If the length of the name is greater than 2 and the name contains only alphabets, the function returns True; otherwise, it logs an error and raises an exception.

```
def is_valid_name(name):
    """
    This function is for validating name that should contain only char,len>2,removing
    :param name: str
    :return: bool
    """
    name = name.replace(".", "").replace(" ", "").replace("_", "")
    if len(name) > 2:
        pass
```

```
    else:
        log.error(f"User name cannot be {len(name)} character")
        raise ValueError(f"Error:-User name cannot be {len(name)} character")

    if name.isalpha():
        pass
    else:
        log.error(f"User name {name} must be str only user ")
        raise ValueError(f"Error:-User name {name} must be str only")
    return True
```

is_valid_dob function is used to validate whether the provided dob (date of birth) is in the format "YYYY-MM-DD". If it is, the function returns True; otherwise, it logs an error and raises an exception.

```
def is_valid_dob(dob, name):
    """
    This function validate the DOB in format="%Y-%m-%d"
    :param dob: str
    :param name: name
    :return: bool
    """
    Format = "%Y-%m-%d"
    try:
        res = datetime.strptime(dob, Format)
        return True
    except ValueError:
        log.error(f'User={name},has wrong format of DOB')
        raise ValueError(f'Error:-User={name},has wrong format of DOB')
```

is_valid_gender function is used to validate whether the provided gender is present in the VALID_GENDER variable. If it is, the function returns True; otherwise, it logs an error and raises an exception.

```
def is_valid_gender(gender, name):
    """
    This function is for validate the gender
    :param gender: str    :param name: str
    :return: bool
    """
    if gender in VALID_GENDER:
        return True
    else:
        log.error(f'User={name} has invalid gender = {gender}')
        raise Exception(f'Error:-User={name} has invalid gender = {gender}')
```

is_valid_password function is used to validate the password. if length of the password is greater than 6 and contains one uppercase, lowercase, digit, special character, then function returns True; otherwise, it logs an error and raises an exception.

```
def is_valid_password(password):
    if len(password) < 6:
        log.error(f"Length of password should not be less than 6")
        raise Exception(f"Length of password should not be less than 6")
```

```
    patterns = {
        "uppercase": r'[A-Z]',
        "lowercase": r'[a-z]',
        "digit": r'\d',
        "special": r'\W'
    }

    for name, pattern in patterns.items():
        if not re.search(pattern, password):
            log.error(f"Password {password} doesn't contain {name} character")
            raise Exception(f"Password {password} doesn't contain {name} character")

    return True
```

is_valid_record function is used to validate the record parameter before inserting it into DATA. It checks four validations in the record data: is_valid_password, is_valid_name, is_valid_dob, and is_valid_gender. If all these functions return True, the record data can be inserted into DATA; otherwise, an exception is raised

```
def is_valid_record(record):
    """
    This function is for validating a new record for inserting in to DATA
    :param record: dict
    :return: bool
    """

    if is_valid_name(record["name"]):
        if is_valid_dob(record["dob"], record["name"]):
            if is_valid_gender(record["gender"], record["name"]):
                if is_valid_password(record["password"]):
                    return True
```

authenticate_user function is used to authenticate the user with his name. If name is in USERS or ADMINS variable, then function returns True; otherwise returns False and it logs an error and raises an exception.

```
def authenticate_user(name):
    """
    :param name:
    :return:
    """
    if name in USERS or name in ADMINS:
        log.info(f"User={name} authentication successfull")
        return True
    return False
```

- **authenticate** function is a decorator that wraps another function to add an authentication check.
- The **wrapper** function takes a name parameter and calls authenticate_user(name). If the user is not authenticated, it raises a PermissionError, otherwise calls the original function.

```
def authenticate(fun):
    def wrapper(name):
        if not authenticate_user(name):
            raise PermissionError(f"Unauthorised user {name} detected..")
```

```
            fun(name)

        return wrapper
```

create_user_info function is used to create a user information that collects input for a user's username, name, dept , dob , gender , password and returns this information in a dictionary.

```
def create_user_info(role):
    global admin
    user = input(f"Enter username:")
    name = input(f"Enter Name:")
    initial = input(f"Enter first character of surname:")
    dept = input(f"Enter the department:")
    dob = input(f"Enter the DOB:")
    gender = input(f"Enter the gender:")
    password = input(f'Enter the password that contains one Upper,lower,digit,special
    if role == "normal":
        admin = False
    elif role == "admin":
        admin = True
    return {"username": user, "name": name + "." + initial, "dept": dept, "dob": dob,
            "password": password, "isadmin": admin}
```

restart function asks the user if they want to restart the program ('Y' for yes, any other key for no). It logs appropriate messages based on the user's choice and returns True if the user chooses to restart, otherwise False.

```
def restart(options):
    """
    These function is used restart the program
    :param options: int
    :return: bool
    """
    restart = input(f"You want to restart(Y/N) = ").upper()
    if restart == "Y":
        log.info(f'Restarting the program....')
        return True
    else:
        log.info(f'Entered Option={options} "Exit"')
        log.info(f'Exiting the program')
        return False
```

APPS/APP.PY

Importing all constants variables in constants module and functions in utility module, email operation and logging logic.

```
from log.logging_logic import *
from emails.email_operations import *
from utilis.utility import *
from constants import *
```

create_user function takes name as parameter and it's designed to facilitate user creation for Authentication and Authorization. @authenticate decorator is used to authenticate the user for create_user function. It verifies if the user has administrative

privileges by checking in the ADMINS list .Upon successful authorization, the function asks for the user's role, either 'normal' or 'admin', and validates this choice. If a valid role is chosen, it generates user information using **create_user_info**, validates the record using **is_valid_record**, logs the creation event, updates ALL_USERS and DATA['records'], sends an email notification to specified recipients, and confirms the addition of the new user record.

```
@authenticate
def create_user(name, role=None):
    """
    These function is used to create user for Authentication and Authorisation
    :param name: str    :param role: str
    :return: bool
    """
    log.info(f'create_user function has started...')
    try:
        # Authorisation
        if name in ADMINS:
            role = input(f"Options are normal, admin:")

            if role == 'normal':
                user_info = create_user_info(role=role)
                if is_valid_record(user_info):
                    log.info(f'Admin {name} has created the normal user')
                    ALL_USERS.append(user_info["name"])
                    DATA["records"].append(user_info)
                    send_email([members for members in receivers],
                               RECORD_ADDED.format(name, user_info["name"], user_info)
                    print(f"New User={user_info["name"]} record added..")

            elif role == 'admin':
                user_info = create_user_info(role=role)
                if is_valid_record(user_info):
                    log.info(f"Admin {name} has created the admin user")
                    ALL_USERS.append(user_info["name"])
                    DATA["records"].append(user_info)
                    send_email([members for members in receivers],
                               RECORD_ADDED.format(name, user_info["name"], user_info)
                    print(f"New User={user_info["name"]} record added..")

            else:
                raise ValueError(f"Incorrect role choosen - available options are norm

        else:
            log.error(f"User={name} does not have create user profile permission")
            raise PermissionError(f"User {name} does not have create user profile perm
    except ValueError as err:
        log.error(err)
        log.info(f'create_user function has ended...')

    except PermissionError as err:
        log.error(err)
        log.info(f'create_user function has ended...')

    except Exception as err:
        log.error(err)
```

```
        log.info(f'create_user function has ended...')
```

Update_record function takes the name as a parameter and is designed to update all user records in DATA.It asks administrator to enter the name of the user whose record needs to be changed and to verify that they are in ALL_USERS.If there , it retrieves their current user record from the DATA, triggering updated information including username, name, gender, and department. The program uses specific validation functions (**is_valid_name** and **is_valid_gender**) to validate these inputs. When these validations return True, it updates the corresponding record in the DATA, logs the change that occurred, sends an email notification to the specified recipients using send_email, and returns an updated DATA.

```
@authenticate
def update_record(name):
    """
    This function update full records in DATA
    :param name: str    :return DATA: dict
    """
    try:
        Name = input(f"Enter the name of user, to modify his record:-")
        log.info(f'Admin has entered name is {Name}')
        if Name in ALL_USERS:
            for item in range(len(DATA["records"])):
                log.info(f'update_record function has started...')
                current_record = DATA["records"][item]
                if Name == current_record["name"]:

                    log.info(f'"message:-Modifying User={Name} record"')
                    print(f'Updating User={Name}  record:-')
                    Username = input(f"Enter the username for User={Name}  record :- '
                    Name = input(f"Enter name for User={Name}  record:- ")
                    Gender = input(f'Enter the gender for User={Name}  record:-')
                    Dept = input(f'Enter your Depeartment fro User={Name}  record:-')

                    log.info(f'"message:- Previous record of User={Name}  is :-" {DATA

                    if is_valid_name(Name):
                        if is_valid_gender(Gender, Name):
                            current_record["username"] = Username
                            current_record["name"] = Name
                            ALL_USERS.append(Name)
                            current_record["gender"] = Gender
                            current_record["dept"] = Dept
                            log.info(f'"message:-" "Modified User={Name}  record is ":
                            send_email([members for members in receivers],
                                       UPDATE_MESSAGE.format(name, Name, DATA["records
                            log.info(f'updated record is = {DATA}')
        else:
            log.error(f'Entered user={Name} has no records')
            print(f'Entered user={Name} has no records')

    except ValueError as err:
        log.error(err)
        log.info(f'update_record function has ended...')

    except Exception as err:
```

```
            log.info(f'update_record function has ended...')
            log.error(err)
        log.info(f'update_record function has ended...')
        return DATA
```

partial_update_record function asks administrator to enter the name of the user whose record wants to update the name field in DATA records by iterating through each record in the data. The is_valid_name function is used by this function to validate the name field. If the function returns True, the name field will be updated and sends an email notification to the specified recipients using send_email function, and returns an updated DATA; if not, an exception is raised.

```
@authenticate
def partial_update_record(name):
    """
    This function is used for partial update of records in DATA
    :param name: dict    :return DATA:  dict
    """
    try:
        Name = input(f"Enter the name of user, to partially update his record = ")
        if Name in ALL_USERS:
            for item in range(len(DATA["records"])):
                if Name == DATA["records"][item]["name"]:
                    log.info(f'For User={Name} --> partial_update_record function has
                    log.info(f'"message:-" "Partial Modifying "{Name}" record"')
                    print(f'Partially updating "{Name}" record:-')

                    Name = input(f'Enter name for "{Name}" record:- ')
                    if is_valid_name(Name):
                        DATA["records"][item]["name"] = Name
                        ALL_USERS.append(Name)
                        send_email([members for members in receivers],
                                   PARTIAL_UPDATE.format(name, Name, DATA["records"][i
                        log.info(f'"message:-" "Partial Modified "{Name}" record is ":

        else:
            log.error(f'Entered user={Name} has no records')
            print(f'Entered user={Name} has no records')

    except Exception as err:
        log.error(err)
        log.info("partial_update_record function has ended...")

    log.info(f"Saved Record={DATA}")
    log.info("partial_update_record function has ended...")
    return f"partially updated saved record are={DATA}"
```

reset_password function, decorated with @authenticate, allows an administrator to reset a user's password by giving the user's name and validating whether user is present in ALL_USERS or not. After verification, the function asks to give a new password that meets specific criteria (uppercase, lowercase, digit, special character). After validating the new password using is_valid_password, it updates the user's password in DATA["records"], logs the password reset event, sends a notification email using send_email, and records the password change in the log.

```
@authenticate
```

```
def reset_password(name):
    """
    These function is used to reset the user password
    :param name: str
    """
    log.info(f'reset_password function has started...')
    try:
        Name = input(f'Enter the name of User,to reset his password = ')
        log.info(f'Admin enterd user name is "{Name}"')
        if Name in ALL_USERS:
            for item in range(len(DATA["records"])):
                if Name == DATA["records"][item]["name"]:
                    Password = input(
                        "Enter the password that should contain one uppercase,lowercas
                    log.info(f'Admin changed user={Name} password to "{Password}"')
                    if is_valid_password(Password):
                        log.info(
                            f'Before the reset password User={Name} password is "{DATA
                        DATA["records"][item]["password"] = Password
                        send_email([members for members in receivers],RESET_PASSWORD.f
                        log.info(
                            f'After the reset password User={Name} password is "{DATA[
        else:
            print(f"User={Name} record is not there")
            log.warning(f"User={Name} record is not there")
    except Exception as err:
        log.error(err)
        log.info(f'reset_password function has ended...')
    log.info(f'reset_password function has ended...')
```

update_user_record function, annotated with @authenticate, allows a user to update their record. The function iterate through each record and it verifies the user's data in DATA["records"] to update username, gender, and department. It validate the data using is_valid_gender if it returns True,:sends a notification email using send_email, and logs the record update process.otherwise raise the excemption and log the error message.

```
@authenticate
def update_user_record(name):
    try:
        log.info(f"update_user_record function has started....")
        for item in range(len(DATA["records"])):
            if name == DATA["records"][item]["name"]:
                log.info(f'user={name} data is in records')
                print(f'{name} has updating his record')

                current_record = DATA["records"][item]
                Username = input(f"Enter the username for {item + 1} record :- ")
                Gender = input(f'Enter the gender for {item + 1} record:-')
                Dept = input(f'Enter your Department for {item + 1} record:-')

                log.info(f'Before updating the record = {DATA}')

                if is_valid_gender(Gender, name):
                    current_record["username"] = Username
                    current_record["gender"] = Gender
```

```
                    current_record["dept"] = Dept
                    log.info(f'After user updated his record = {DATA}')
                    send_email([members for members in receivers],
                               UPDATE_USER_RECORD.format(name,DATA["records"][item]))
                    print(f'Updated user={name} record = {DATA["records"][item]}')
    except ValueError as err:
        log.error(err)
        log.info(f"update_user_record function has started....")

    except Exception as err:
        log.error(err)
        log.info(f"update_user_record function has started....")
    log.info(f'update_user_record function has ended...')
    return DATA
```

user_info function is used to check the user information in DATA. if the authenticator is admin, can view the any user information and if authenticator is normal user, can view only his information. After viewing the information it sends a notification email using send_email, log message will be logged. If specific user name is not in records it raise a exception and error log message will be logged.

```
@authenticate
def user_info(name):
    """
    These function is about checking user detailed information.
    if user is admin,can see any user information
    if normal user,can see only his information
    :param name: str
    :return: dict
    """
    log.info(f'user_info function started...')
    try:
        if name in ADMINS:
            Name = input("Enter the name of user to see his details:- ")
            log.info(f'Admin={name} has entered "{Name}" to see his information')
            if Name in ALL_USERS:
                for item in range(len(DATA["records"])):
                    if Name == DATA["records"][item]["name"]:
                        print(f"User={Name} information :- {DATA["records"][item]}")
                        send_email([members for members in receivers],
                                   CHECK_USER_INFO.format(name,Name))
                        log.info(f"User={Name} information :- {DATA["records"][item]}'
            else:
                log.error(f"User={Name},is not in records")
                raise Exception(f"User={Name},is not in records")

        if name in USERS:
            log.info(f"User={name} having checking his information")
            for item in range(len(DATA["records"])):
                if name == DATA["records"][item]["name"]:
                    print(f"Your information :- {DATA["records"][item]}")
                    send_email([members for members in receivers],
                               CHECK_USER_INFO.format(name, Name))
                    log.info(f"Your information :- {DATA["records"][item]}")
    except Exception as err:
```

```
        log.error(err)
        log.info(f'user_info function ended...')
    log.info(f'user_info function ended...')
```

delete_record function, annotated with @authenticate decorator, deletes a user's record from DATA and ALL_USERS. the function asks the user's name to enter, if user name in ALL_USERS, removes their record from DATA["records"], sends a notification email using send_email, logs the deletion event, and handles exceptions if the user does not exist.

```
@authenticate
def delete_record(name):
    """
    This function will delete the record in DATA
    :param name: dict    :return DATA:  dict
    """
    log.info(f'delete_record function has started...')
    try:
        Name = input("Enter the name of user:- ")
        if Name in ALL_USERS:
            for item in range(len(DATA["records"])):
                if Name == DATA["records"][item]["name"]:
                    del DATA["records"][item]
                    ALL_USERS.remove(Name)
                    send_email([members for members in receivers],
                               DELETE_MESSAGE.format(name,Name))
                    log.info(f'Deleted the {Name} record...')
                    print(f"Deleted the {Name} record...")
                    break
        else:
            raise Exception(f'user {Name} is not there...')

    except Exception as err:
        log.error(err)
        log.info(f'delete_record function has ended....')
    log.info(f'delete_record function has ended....')
```

read_records function, annotated with @authenticate decorator. It shows the all user records in DATA. sends a notification email using send_email, log message will be logged.

```
@authenticate
def read_records(name):
    print(f'Saved Record = {DATA}')
    send_email([members for members in receivers], GET_ALL_MESSAGE.format(name))
    log.info(f'Saved Record = {DATA}')
```

restart_program function takes the name of the user, if user is admin; he has following function properties to access there are :- create_user, update_record, partial_update_record, user_info, delete_record, reset_password, read_records. if user is normal, having these function properties create_user,update_user_record,user_info,read_records.

```
def restart_program():
    name = input("Enter the name of user :-")
```

```
    log.info(f"Entered name of user is = {name}")
    if name in ADMINS:
        log.info(f"User={name} is an admin, having Admin properties")
        print(f"User={name} is an admin, having Admin properties")
    elif name in USERS:
        print(f"User={name} is a user, having User properties.")
        log.info(f"User={name} is a user, having User properties")
    else:
        log.warning(f"Unauthorized User={name}, trying to access the application...")
        send_email([members for members in receivers], UNAUTHENTICATED_MESSAGE.format(
        print(f"Email is sent to respective team members. \n Unauthorized User={name},
        exit()

    while True:
        if name in ADMINS:
            options = int(input(ADMIN_CONSOLE))
            if options == 1:
                log.info(f'Entered Option=1 "Create User"')
                create_user(name)
            elif options == 2:
                log.info(f'Entered Option=2 "Update Record"')
                print(update_record(name))

            elif options == 3:
                log.info(f'Entered Option=3 "Partially update record"')
                print(partial_update_record(name))
            elif options == 4:
                log.info(f'Entered Option=4 "Check specify user info"')
                user_info(name)
            elif options == 5:
                log.info(f'Entered Option=5 "Delete Record"')
                delete_record(name)
            elif options == 6:
                log.info(f'Entered Option=6 "Reset User Password"')
                reset_password(name)
            elif options == 7:
                log.info(f'Entered Option=6 "Read Record"')
                read_records(name)
            elif options == 8:
                if restart(options):
                    restart_program()
                else:
                    exit()
            else:
                log.info(f'Entered wrong Option={options}')
                print(f"Choosed wrong option ={options}")
                options = int(input(ADMIN_CONSOLE))

        elif name in USERS:
            options = int(input(USER_CONSOLE))
            if options == 1:
                log.info(f'Entered Option=1 "Create User"')
                create_user(name)
            elif options == 2:
                log.info(f'Entered Option=2 "Update your record"')
```

```
                update_user_record(name)
            elif options == 3:
                log.info(f'Entered Option=3 "Check Your Details"')
                user_info(name)
            elif options == 4:
                log.info(f'Entered Option=4 "Read record"')
                read_records(name)

            elif options == 5:
                if restart(options):
                    restart_program()
                else:
                    exit()
            else:
                log.info(f'Entered wrong option={options}')
                print(f'Choosed wrong option ={options}')
                options = int(input(USER_CONSOLE))
        else:
            break


restart_program()
```