App Development Documentation for User Management System

Introduction

THIS DOCUMENT PROVIDES AN OVERVIEW OF THE USER MANAGEMENT SYSTEM APP, FUNCTION DESCRIPTIONS, CODE, CODE EXPLANATION, USAGE EXAMPLES, AND LOGGING CONFIGURATION.

constants.py

- This module is for constants which contains Data Base related data.
- Added 6 constants :- DATA, AVAILABLE_RECORDS, VALID_COUNTRY_LIST, EXCLUDED_NUMBERS ,USERS,ADMINS

```
# DATABASE RECORD
DATA = {"records": [{"mobile": 914234234245, "name": "Kumar Makala", "company": "KXN"}
                    {"mobile": 915421215452, "name": "Komal", "company": "APPLE"},
                    {"mobile": 913020022100, "name": "Mahesh", "company": "MICROSOFT"}
                    {"mobile": 910000000000, "name": "Rajesh", "company": "WIPRO"},
                    {"mobile": 914111111111, "name": "Suresh", "company": "MIND TREE"}
```

```
AVAILABLE_RECORDS = [item["mobile"] for item in DATA["records"]]
```

```
VALID_COUNTRY_LIST = ["91", "45", "67", "56"]
```

```
EXCLUDED_NUMBERS = [9898989898, 9999999999, 8888888888]
```

```
USERS = ['kmakala', 'dinesh', 'kmahesh']
```

```
ADMINS = ['komal']
```

logging.py

This file contains logging configurations.

```
import logging

logging.basicConfig(level=logging.DEBUG,
                    format='%(asctime)s - %(levelname)s - %(message)s',
                    filemode='a', filename='main.log')
```

authentic_authorised.py

This contains Admins and Users related data, who can access the codes.
we need to import required modules as follows :

```
from app_development.app.constants import USERS, ADMINS
from app_development.logging_activity.logging_module import logging
```

DECORATOR FUNCTION :

This decorator function takes authenticate user function, checks if not authenticate_user(name) then it alerts the users for unrestricted entries by sending red flags message to the users.

```
def authenticate_decorator(fun):
    """
    This is decorator func
    :param fun: function
    :return: bool
    """

    def wrapper(name, *args, **kwargs):
        logging.info(f"Decorator Function Entered")
        try:

            if not authenticate_user(name):
                logging.info(f"Red Alert.....This is user not in Admin list {ADMINS}..
                print(f"Red Alert.....This is user not in Admin list {ADMINS}")
                print(f"Un Restricted Entry")
                logging.error(f"Hey unrestricted entry noted with name - {name}")
                raise PermissionError(f"Hey unrestricted entry noted with name - {name
            logging.info(f"Decorator Function Ended successfully \n")
            return fun(name, *args, **kwargs)
        except Exception as err:
            logging.error(f"Exception is {err}")
            return err

    return wrapper
```

AUTHENTICATE_USER(NAME) FUNCTION :

This takes name as parameter and checks if the name is in users or admins.

```

```

```
def authenticate_user(name):
    """
    Checks who is the person
    :param name: str
    :return: bool
    """
    logging.info(f"authenticate_user Function Entered")
    if name in USERS or name in ADMINS:
        logging.debug(f"User {name} authenticated successfully - meaning basic info ve
        return True
    logging.info(f"authenticate function ended")
    return False
```

FOR_CREATE_DELETE_USER_CONDITIONS(NAME) FUNCTION :

This is used with decorator function, we denote @authenticate_decorator above the function to link with the decorator.This function takes name as parameter and checks for only the user is strictly an admin or not. If the role is normal user then he has no permission to create or delete the other user information . Only Admin can have this privelage .

```
@authenticate_decorator
def for_create_delete_user_conditions(name):
    """
    This function is used for giving permission to user for create, delete
    :param name: str
    :return: bool
    """

    # Authorisation

    role = input(f"Options are n, a:")

    if role == 'n':
        print(f'(" Normal USER Role ")')
        print(f"Normal User {name}... so, doesn't have permission to create new record
        logging.error(f"Normal User {name}... so, doesn't have permission to create ne
        raise PermissionError(f"Normal User {name}... so, doesn't have permission to c
    elif role == 'a':
        print(f'(" ADMIN Role ")')
        print(f"Admin User {name}... so, have permission to create new record")
        return True
    else:
        raise ValueError(f"Incorrect role chosen - available options are 'n' and 'a' c
```

FOR_UPDATE_READ_USER_CONDITIONS(NAME) FUNCTION :

This is used with decorator function. This function takes name as parameter and checks whether user or admin can perform updation , read the codes and no other users has permission to access this.

```
@authenticate_decorator
def for_update_read_user_conditions(name):
    """
        This function is used for giving permission to user for update, read
        :param name: str
        :return: bool
    """
    role = input(f"Options are n, a:")

    if role in ['n', 'a']:
        logging.debug(f"check if user selected either n or a ")
        return f"  Cool.....Authorised user only..... "
    else:
        logging.error(f"Incorrect role chosen - available options are 'n' and 'a' only
        raise ValueError(f"Incorrect role chosen - available options are 'n' and 'a' c
```

CREATE_ADMIN_NORMALUSER_INFO(ROLE) FUNCTION :

This is used with decorator function.It takes role as an input parameter and checks if the name is matched within the existing user or admin names if not then it will create new user details and in this we can provide Admin or Normal user role to the new entry one.

```
@authenticate_decorator
def create_admin_normaluser_info(role):
    """
    This function is used to create/add new admins as well as new normal users into AD
    :param role: str
    :return: dict
    """
    try:
        logging.info(f"create admin normal user info function entered\n")
        user = input(f"Enter username:")
        if not user:
            logging.error(f"Username cannot be empty.")
            raise ValueError("Username cannot be empty.")
        # Mandatory
        firstname = input(f"Enter First name:")
        if not firstname:
            logging.error(f"First name cannot be empty.")
            raise ValueError("First name cannot be empty.")
        middlename = input(f"Enter middle name:")  # Optional
        lastname = input(f"Enter last name:")  # Optional
        dept = input("Enter the department: ")
        if not dept:
```

```
            logging.error(f"Department cannot be empty.")
            raise ValueError("Department cannot be empty.")
        dob = input(f"Enter the DOB:")
        if not dob:
            logging.error(f"DOB cannot be empty.")
            raise ValueError("DOB cannot be empty.")

        if role == "n":
            admin = False
            USERS.append(user)
        elif role == "a":
            admin = True
            ADMINS.append(user)
        else:
            logging.error(f"Invalid role. Use 'n' for normal user or 'a' for admin.")
            raise ValueError("Invalid role. Use 'n' for normal user or 'a' for admin.'
        logging.info(f"{user} created in create_admin_normaluser_info function")
        return {"username": user, "name": firstname + middlename + lastname, "dept": d
    except ValueError as e:
        logging.error(e)
        return {"error": str(e)}
```

email_config.py

In this we should provide the basic setup for email generations like SMTP_PORT number, SMTP_SERVER ,SENDER_EMAIL, RECIEVER_EMAIL, PASSWORD, MESSAGE.

- we need to setup password for sending messages. For setup go to → open google → manage your personal google account → search for app passwords →i f not found then u have to setup 2step verification to ur email id. Then it will works. After goto → app passwords and it will asks for app name, give name of ur wish and then click create. Then u can see the 16 characters password copy it and paste it in PASSWORD as shown in below code.

```
SMTP_PORT = 587  # For starttls
SMTP_SERVER= "smtp.gmail.com"
SENDER_EMAIL = "komalsaikiran05@gmail.com"
RECEIVER_EMAIL = ["komalsaikiran05@gmail.com"]
PASSWORD = "qlqgqoyzaynbogra"
MESSAGE = """\
Subject: Hi there

New User has been Posted successfully in your table ."""
DELETE_MESSAGE = """
User has been deleted successfully from your table

"""
UPDATE_MESSAGE = """
User has been updated successfully

"""
GET_ALL_MESSAGE = """
Some one is checking all users information from the table
```

```
"""
GET_SINGLE_MESSAGE = """
Some one is checking single user information from the table
"""
GET_USER_RANGE = """
Some one is checking User Range information from the table
"""
print("This is smtp_port---", SMTP_PORT)
print("This is email_password", PASSWORD)
```

email_constants.py

We need to import the following :

```
import smtplib
import ssl
from app_development.email_setup.email_config import *
```

SEND_EMAIL(TOO_EMAIL=NONE, EMAIL_BODY="") FUNCTION :

This function is used to send emails when ever there is changes made in CRUD operations.

```


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


def send_email(too_email=None, email_body=""):
    """
    This function is used to send emails when ever there is changes in CRUD operations
    :param too_email: list of email addresses needed to be sent
    :param email_body: The message which user needs to be notified
    :return: None
    """
    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)
```

user_utils.py

Here required constants are imported and this module contains utilities(codes) which are repeated. Programmers can use these utilities according to their needs.

Imports :

```
from App_Development.app.constants import *
from App_Development.logging_activity.logging_module import *
```

Code 1 :

```
def is_valid_mobile(mobile):
    """
    This function checks whether the mobile number is valid or not.
    :param mobile: int
    :return: bool
    """
    logging.info(f" ------is valid mobile function entered------\n")
    try:
        if isinstance(mobile, int):
            converted_str = str(mobile)
            if len(converted_str) == 12:
                if converted_str[:2] in VALID_COUNTRY_LIST:
                    logging.debug(f"converted str {converted_str[:2]} is in {VALID_COU
                    logging.info("------is valid mobile function ended------")
                    return True
                else:
                    logging.error(f"Invalid country code - {converted_str[:2]} which i
                    raise ValueError(f"Invalid country code - {converted_str[:2]} whic
            else:
                logging.error(f"Invalid length of mobile - {converted_str}, should be
                raise ValueError(f"Invalid mobile number length - {mobile}")
        else:
            logging.error(f"Invalid mobile type - {mobile}, should be an integer")
            raise ValueError(f"Invalid mobile number type - {type(mobile)}")
    except Exception as err:
        return err
```

Code 1 Explanation :

Checks whether the mobile is valid or not ...try block enters -->> isinstance(mobile, int) which checks that

mobile number provided by the user is integer or not and then converts it to string to perform the string methods as it was easy to handle and perform string methods. Also checks length of the mobile number equals to 12 or not and after it checks If converted string[:2] which means starting of the mobile number starts with in the country codes provided in VALID_COUNTRY_LIST in constants.py module. Return True if it presents or raise the error. And every moment is captured using log method.....Also used used exception handling for calling this function at the end inorder to save it from crashing of the code

Example Usage

```
try:
    mobiles = 912459878954
    is_valid_mobile(mobiles)
    logging.info(f"Result: {mobiles}")
except ValueError as e:
    logging.error(e)
```

Code 2 :

```
def is_excluded(mobile_num):
    """
    This function checks whether the mobile number is in the exemptions mobile number
    :param mobile_num: int
    :return: bool
    """
    logging.info(f" ------is excluded function entered------\n")

    if not isinstance(mobile_num, int):
        logging.error(f"Invalid type for mobile_num: {type(mobile_num)}. Expected type
        raise ValueError(f"Invalid type for mobile_num: {type(mobile_num)}. Expected t

    if mobile_num in EXCLUDED_NUMBERS:
        logging.debug(f"mobile_num {mobile_num} is in EXCLUDED_NUMBERS {EXCLUDED_NUMBE
        return True
    logging.warning(f"mobile_num {mobile_num} is not in {EXCLUDED_NUMBERS}.")
    return False
```

Code 2 Explanation :

This is used to take user mobile number as an input and checks if the mobile number is int or not and if it is not integer it will raise an error saying that it is not int type and if it is integer then checks mobile number in EXCLUDED_NUMBERS which is declared in constants and it is mainly for VIP numbers which the validations shouldnot be performed. It will returns True if it is in EXCLUDED_NUMBERS , or it will shows an error returning False. Also used used exception handling for calling this function at the end in order to save it from crashing of the code.....Also used used exception handling for calling this function at the end in order to save it from crashing of the code

Example Usage

```
try:
    mobile_nums = 912459878954
    is_excluded(mobile_nums)
    logging.info(f"Result: {mobile_nums}")
except ValueError as e:
    logging.error(e)
```

Code 3 :

```
def is_valid_country(converted_str):
    """
    This function checks whether the mobile number matches the given country code or r
    :param converted_str: str
    :return: bool
    """
    logging.info(f" ------is valid country function entered------\n")
    if converted_str[:2] in VALID_COUNTRY_LIST:
        logging.debug(f"converted_str {converted_str} is in VALID_COUNTRY_LIST {VALID_
        return True
    else:
        logging.error(f"Invalid country code - {converted_str[:2]} which is not in {VA
        raise ValueError(f"Invalid country code - {converted_str[:2]}. Valid country c
```

Code 3 Explanation :

This takes converted string for example converted str of mobile "561245625142", and checks if the first 2 digits are in the VALID_COUNTRY_LIST which are declared in the constants module. If it is Valid then returns True or else will return error which can be captured using log method .....Also used used exception handling for calling this function at the end in order to save it from crashing of the code

Example Usage

```
try:
    converted_strs = "561245625142"
    is_valid_country(converted_strs)
    logging.info(f"Result: {converted_strs}")
except ValueError as e:
    logging.error(e)
```

Code 4 :

```
def is_mobile_length_valid(converted_str):
    """
    This function checks whether the mobile number length matches the desired length c
    :param converted_str: str
    :return: bool
    """
    logging.info(f" ------is mobile length valid function entered------\n")
    if len(converted_str) == 12:
        logging.debug(f"converted_str {converted_str} is of valid length (12 digits)")
        logging.info("------is mobile length valid function ended------")
        return True
    else:
        logging.error(f"Invalid mobile length - {converted_str}, should be length of 1
        raise ValueError(f"Invalid mobile number length {len(converted_str)}. Valid le
```

Code 4 Explanation :

This takes converted str as input parameter for example "561245625142" which is given by user and checks if length of the mobile(converted str) is equal to 12. If the length matches then it captures the success message and if not it will captures the error stating that mobile number is not valid one.....Also used used exception handling for calling this function at the end in order to save it from crashing of the code

Example Usage

```
try:
    converted_strs = is_mobile_length_valid("564215210212")
    logging.info(f"Result: {converted_strs}")
except ValueError as e:
    logging.error(e)
```

Code 5 :

```
def is_valid_type(mobile):
    """
    This function checks whether the type of the mobile number is valid or not.
    :param mobile: int
    :return: bool
    """
    logging.info(f" ------is  valid type function entered------\n")
    if isinstance(mobile, int):
```

```
        logging.debug(f" {mobile} is of valid type -- {isinstance(mobile, int)}")
        return True
    else:
        logging.error(f"Invalid mobile type - {mobile}, should be an integer")
        raise ValueError(f"Invalid mobile number type - {type(mobile)}")
```

Code 5 Explanation :

> This takes user provided mobile number as input parameter and checks whether the mobile number type is of integer or not. If it is integer then it will return True or it will be captured as invalid mobile type. For example user sends “kjajc” which shows error and the correct one is 91244210022 this type.....Also used used exception handling for calling this function at the end in order to save it from crashing of the code

Example Usage

```
try:
    mobiles = is_valid_type(561245789654)
    logging.info(f"Result: {mobiles}")
except ValueError as e:
    logging.error(e)
```

Code 6 :

```
def is_valid_record(RAW_DATA, record):
    """
    This function checks whether the record is valid or not.
    :param RAW_DATA: dict
    :param record: list of dict
    :return: bool
    """
    logging.info(f" ------is  valid record function entered------\n")

    if not isinstance(RAW_DATA, dict):
        logging.error(f"Invalid data structure - {RAW_DATA}, should be a dict")
        raise ValueError(f"Invalid input DATA {RAW_DATA}")

    if not isinstance(record, dict):
        logging.error(f"Invalid data structure - {record}, should be a dict")
        raise ValueError(f"Record should be a dictionary, got {type(record)} instead.'

    if "mobile" not in record:
        logging.error(f"Missing 'mobile' key in record: {record}")
        raise ValueError(f"Missing 'mobile' key in record: {record}")

    logging.debug(f"Record {record} is valid")
```

```
    return True
```

Code 6 Explanation :

> This takes RAW_DATA, record lets say.....(RAW_DATA - dictionary { }, and record - list of dictionaries [{ }]) as input parameters and checks whether RAW_DATA is dictionary or not and also checks record is also dictionary or not. If both are dictionaries then it will return and captures as True or it will raise an error. And also checks the key -mobile is not in record, if not in record then it will throws an error ...Also used used exception handling for calling this function at the end in order to save it from crashing of the code

Example Usage

```
try:
    valid_record = is_valid_record(DATA, {"mobile": 9898989898, "name": "Suresh", "com
    logging.info(f"Result: {valid_record}")
except ValueError as e:
    logging.error(e)
```

main.py

Imports from required modules are in below:

```
from app_development.utils.user_utils import *
from app_development.app.constants import *
from app_development.logging_activity.logging_module import logging
from app_development.email_setup.email_constants import *
from app_development.authentication_authorisation.aunthentic_authorised import (
    authenticate_decorator, for_create_delete_user_conditions, for_update_read_user_co
```

`NEW_RECORD(NAME, RECORD) - FUNCTION  :-`

This function inserts a new record into the data storage.

Parameters :

- name : str
- `record (dict)`: It takes dictionary containing user details with at least a `mobile` key.

Returns :

- `dict`: A dictionary with a success message and the inserted record, or an error message.

Code :

```
@authenticate_decorator
# POST METHOD
def new_record(name, record):
    """
        This method inserts new record.
        :param name: str
        :param record: list of dictionary
        :return: dict
    """
    logging.info(f"--------------New Record function Entered  -----------------\n")

    try:

        if for_create_delete_user_conditions(name):

            # Check if 'mobile' key exists in record and is valid
            if "mobile" not in record:
                logging.error(f"Missing mobile key in the record {record}")
                raise ValueError("Missing 'mobile' key in record")
            mobile_number = record["mobile"]
            logging.debug(f" mobile number is extracted - {mobile_number}")

            # Check if mobile number already exists in AVAILABLE_RECORDS list
            if mobile_number in AVAILABLE_RECORDS or mobile_number in EXCLUDED_NUMBERS
                logging.warning(f"Record with mobile number {mobile_number} already ex
                return "Duplicate Record Found"

            # Insert record into DATA and update AVAILABLE_RECORDS list
            DATA["records"].append(record)
            AVAILABLE_RECORDS.append(mobile_number)

            send_email(["komalsaikiran05@gmail.com"], MESSAGE)
            print(f"Email has been sent to the users ")

            logging.debug(f"Record inserted successfully. New record added to AVAILABL

            logging.info(f"Record successfully inserted using POST METHOD: {record}")
            return {"message": "Successfully inserted into the record", "record": reco
        else:
            return {"error": "No user details found"}
    except KeyError as err:
        error_message = f"KeyError: {err}. Please check the structure of the record."
        logging.error(error_message)
        return {"message": error_message, "status": "Failed"}

    except ValueError as err:
        error_message = str(err)
```

```
        logging.error(f"Error inserting record: {error_message} using POST METHOD")
        return {"message": error_message, "status": "Failed"}
```

Code Explanation :

Decorator @authenticate_decorator is used in this function and It takes name and dictionary as input parameter and enters into the function. Then it records the entry log and enters try block and if for_create_delete_user_conditions(name) it checks whether the user is Admin or not..if admin then Check if 'mobile' key exists in record or not if it is present then it will raise the value error, or will continue executing next line that is assigning record key “mobile” to mobile number -->>
mobile_number = record["mobile"]
Then checks if mobile number is present in records or not, if present then it raises error. After , DATA["records"].append(record) - which means appending record(which is list of dict) to DATA having records as key , and available_records.append(mobile_number) - which means appending mobile number to available records. If any error occurs it will be captured as we used logging module.

Example Usage

```
calling new record function here :-
```

```
inserted_record = new_record("komal", {"mobile": 935620002021, "name": "Hemanth", "com
logging.info(
    f" {inserted_record} has been inserted using mobile number \n")
logging.info(f"----------------------New Record function Ended-----------------------
```

```
--------------------------------------------------------------------------------
```

`GET_SINGLE_USER_DETAILS(NAME, RECORD) - FUNCTION :-`

This method get details of a single user based on the mobile number.

Parameters

- name : str

- `record (dict)`: A dictionary containing a `mobile` key.

Returns

- `dict`: The user's details if found, otherwise an error message.

Code :

```
# GET METHOD SINGLE USER DETAILS
@authenticate_decorator
def get_single_user_details(name, record):
    """
        This method gets single record.
        :param name: str
        :param record: list of dictionary
        :return: dict
    """
    logging.info(f"-----------GET Single User METHOD Entered  ----------------\n")

    try:

        if for_update_read_user_conditions(name):

            mobile = record["mobile"]
            logging.debug(f"mobile number extracted - {mobile}")
            if is_valid_mobile(record["mobile"]):
                for user in DATA['records']:
                    if user["mobile"] == mobile:
                        logging.info(f"User details found -->> {user}")

                        send_email(["komalsaikiran05@gmail.com"], GET_SINGLE_MESSAGE)
                        print(f"Email has been sent to the users ")

                        return user
                logging.warning(f"No user details found for mobile number in GET METHO
                                f" so GET METHOD will return nothing.")
                # print(f"No user details found for mobile number {mobile}.")
                return {"error": "User details not found "}
            else:
                return {"error": "No user details found"}
    except ValueError as err:
        logging.error(f"Error in GET METHOD: {err}")
        return {"error": str(err)}
```

Code Explanation :

Decorator @authenticate_decorator is used in this function and It takes name and record(list of dictionary) as an input parameters and enters try block with log message.Then
mobile = record["mobile"] which means record having mobile as key is assigned to mobile and it goes for valuation of is valid mobile which it takes record having mobile as key as parameter. Then it enters for loop to check user is in DATA['records'] and if user["mobile"] == mobile then it records that the user is found or it shows the error.

Example Usage

```
calling get_single_user_details(namerecord) here :-

single_user_details = get_single_user_details("kmahesh", {"mobile": 915421215452})
logging.info(
    f" {single_user_details} has been viewed successfully using mobile number {mobiles

logging.info(f"-----------------GET Single User METHOD Ended----------------- \n")
```

`--------------------------------------------------------------------------------`

`GET_ALL_USER_DETAILS(NAME, RAW_DATA)  - FUNCTION :-`

This method retrieves details of all users.

Parameters

- `name  :  str`
- `raw_DATA (dict)`: The raw data dictionary containing all user records.

Returns

- `list`: A list of dictionaries containing all user records, or an error message.

Code :

```
@authenticate_decorator
def get_all_user_details(name, raw_DATA):
    """
        This method inserts new record.
        :param name: str
        :param raw_DATA: dictionary
        :return: list of dict
    """
    logging.info(f"-----------GET All User Details METHOD Entered  --------- \n")
```

```
    try:
        if for_update_read_user_conditions(name):
            # Assuming 'DATA' is a dictionary containing a key 'records' which is a li
            if 'records' in raw_DATA:

                logging.debug(f"All User details are  -->> {raw_DATA['records']}")
                send_email(["komalsaikiran05@gmail.com"], GET_ALL_MESSAGE)
                print(f"Email has been sent to the users ")
                return raw_DATA['records']
            else:
                logging.warning(f"No user details found in GET METHOD.")
                return {"error": "No user details found "}
        else:
            return {"error": "Unauthorized access"}
    except ValueError as err:
        logging.error(f"Error in GET METHOD: {err}")
        return {"error": str(err)}
```

Code Explanation :

Decorator @authenticate_decorator is used in this function and It takes dictionary as an input parameter and enters try block after capturing log. if for_update_read_user_conditions(name) it checks if it got satisified or else will not execute this code.If works Then it checks if records key is present in DATA which is a dictionary, if present then it will return all users which are found in the dictionary DATA. Else it will capture the error.

Example Usage

```
calling get_all_user_details(name, raw_DATA)  :-

all_users = get_all_user_details("komal", DATA)
logging.info(
    f" {all_users} has been displayed here \n")
logging.info(f"-----------------GET All User Details METHOD Ended----------------- \n'
```

`--------------------------------------------------------------------------------`

`DELETE_USER_DETAILS(NAME, RECORD)   - FUNCTION :-`

This method deletes a user's details based on the mobile number.

Parameters

- name : str
- `record (dict)`: A dictionary containing a `mobile` key.

Returns

- `dict`: The deleted user's details if found, otherwise an error message.

Code :

```
@authenticate_decorator
def delete_user_details(name, record):
    """
            This method inserts new record.
            :param name: str
            :param record: list of dictionaries
            :return:  dict
    """
    logging.info(f"-----------------Delete User Details METHOD Entered----------------
    try:
        if for_create_delete_user_conditions(name):

            logging.info(f"Delete user details function started")
            mobile = record["mobile"]
            logging.debug(f"mobile number is extracted - {mobile}")
            if is_valid_mobile(record["mobile"]):
                for x, user in enumerate(DATA['records']):
                    if user["mobile"] == mobile:
                        logging.debug(f"User details found -->> {user}")
                        # print(f"User details found -->> {user}")
                        delete_user = DATA['records'].pop(x)
                        send_email(["komalsaikiran05@gmail.com"], DELETE_MESSAGE)
                        logging.info(f"Email has been sent to the users ")
                        logging.info(f"User deleted successfully in DELETE METHOD: {de
                        return delete_user

                logging.warning(f"No user details found for mobile number {mobile} so
                # print(f"No user details found for mobile number {mobile}.")
                return {"error": "User details not found "}
            else:
                return {"error": "Invalid mobile number"}
        else:
            logging.error(f"Un Authorised Entry detected {name}")
            raise ValueError(f"Un Authorised Entry detected {name}")
    except ValueError as err:
        logging.error(f"Error: {err}")
        return {"error": str(err)}
```

Code Explanation :

Decorator @authenticate_decorator is used in this function and It takes record as parameter and enters the try block capturing the entry by log method. It checks if the user mobile is valid or not if it is valid then it enters

for loop while checking the condition that x (which is index) and user(which is iterable item) is in enumerate(DATA['records']) - which means enumerate generates index and its corresponding value to it from the dictionary. Then if user mobile matches with the mobile which is in dictionary then it will be deeted using pop method. delete_user = DATA['records'].pop(x).

Example Usage

```
calling  delete_user_details(name, record) :-

delete_user_details = delete_user_details("komal", {"mobile": 914111111111})
logging.info(f"User deleted successfully in DELETE METHOD: {delete_user_details} \n ")
logging.info(f"-----------------Delete User Details METHOD Ended----------------- \n")
```

`--------------------------------------------------------------------------------`

`PATCH_USER_DETAILS(NAME, RAW_DATA, RECORD)  - FUNCTION :-`

This method updates a user's details based on the mobile number.

Parameters

- name : str
- `record (dict)`: A dictionary containing the new user details with at least a `mobile` key.
- `raw_DATA (dict)`: The raw data dictionary containing all user records.

Returns

- `dict`: The updated user's details if found, otherwise an error message.

Code :

```
@authenticate_decorator
def patch_user_details(name, raw_DATA, record):
    """
            This method inserts new record.
            :param name: str
            :param record: list of dicts
            :param raw_DATA: dictionary
            :return:  dict
    """
    logging.info(f"----------------PATCH METHOD Entered--------------- \n")
```

```
    try:
        if for_update_read_user_conditions(name):
            mobile = record["mobile"]
            logging.debug(f"mobile number extracted : {mobile}")

            if is_valid_mobile(mobile):
                for user in raw_DATA['records']:
                    if user["mobile"] == mobile:
                        logging.info(f"User found for updating -->> {user}")

                        # Update user details
                        user.update(record)
                        logging.info(f"User details updated to -->> {user}")
                        send_email(["komalsaikiran05@gmail.com"], UPDATE_MESSAGE)
                        print(f"Email has been sent to the users ")
                        return user

                logging.warning(f"No user details found for mobile number in PATCH MET
                return {"error": "User details not found"}

            else:
                logging.error(f"Invalid details found for mobile number in PATCH METHO
                raise ValueError("Invalid mobile number format")
        else:
            return {"error": "Unauthorized access"}
    except ValueError as err:
        logging.error(f"Error in PATCH METHOD: {err}")
        return {"error": str(err)}
```

Code Explanation:

Decorator @authenticate_decorator is used in this function and It takes raw DATA which is dictionary and record which is list of dictionary as input parameters and enters try block capturing the entry.if for_update_read_user_conditions(name) true, Then checks if mobile is valid or not and for valid mobile and for user in raw DATA having records as key, then check if user having mobile as key is equal to mobile or not, if correct then user.update(record). Or it shows error.

Example Usage

```
calling  patch_user_details(name, raw_DATA, record) :-

record_to_update = {"mobile": 914111111111, "name": "Jack"}
updated_user = patch_user_details("komal", DATA, record_to_update)
logging.info(
    f"User partial updation done successfully in PATCH METHOD: {updated_user}, updated
logging.info(f"-----------------Patch User Details METHOD Ended----------------- \n")
```