Single Route

1. OVERVIEW DOCUMENT

Project Overview

This Flask application is designed for profile management. It allows for CRUD operations (Create, Read, Update) on user profiles and handles validation and authorization.

2. API ENDPOINTS DOCUMENTATION

Endpoint: `/api/user/<username>`

- **Description**: Manages user profiles. Supports GET, POST, and PATCH methods.
- **Methods**:
  - **GET**: Retrieve information about a specific user.
  - **POST**: Create a new user profile.
  - **PATCH**: Update an existing user profile.

Request and Response Details:

1. **GET Method**

   - **Description**: Fetches details of a user profile.
   - **Request**:
     - **Method**: GET
     - **URL**: `http://localhost:5000/api/user/<username>`
     - **Headers**:
       - `X-Current-User` (string): Username of the requester.
       - `X-Is-Admin` (boolean): Indicates if the requester has admin privileges.
   - **Responses**:
     - **200 OK**: Successfully retrieved user information.

```
{
"email": "example@example.com",
"age": 30,
"mobile": "1234567890",
"gender": "male",
"blood_group": "O+"
}
```

     - **403 Forbidden**: Requester does not have permission to access the information.
     - **404 Not Found**: User does not exist.

1. **POST Method**

- **Description**: Creates a new user profile.
- **Request**:
  - **Method**: POST
  - **URL**: `http://localhost:5000/api/user/<username>`
  - **Body** (JSON):

```
{
"email": "newuser@example.com",
"age": 28,
"mobile": "9876543210",
"gender": "female",
"blood_group": "A+"
}
```

- **Responses**:
  - **201 Created**: User profile created successfully.

```
{
"email": "newuser@example.com",
"age": 28,
"mobile": "9876543210",
"gender": "female",
"blood_group": "A+"
}
```

  - **400 Bad Request**: Invalid user data or user already exists.

1. **PATCH Method**

- **Description**: Updates an existing user profile.
- **Request**:
  - **Method**: PATCH
  - **URL**: `http://localhost:5000/api/user/<username>`
  - **Headers**:
    - `X-Current-User` (string): Username of the requester.
    - `X-Is-Admin` (boolean): Indicates if the requester has admin privileges.
  - **Body** (JSON):

```
{
"age": 29
}
```

- **Responses**:
  - **200 OK**: User profile updated successfully.

```
{
"email": "existinguser@example.com",
"age": 29,
"mobile": "1234567890",
"gender": "male",
"blood_group": "O+"
}
```

- **403 Forbidden**: Requester does not have permission to update the profile.
- **404 Not Found**: User does not exist.

---

3. CODE DOCUMENTATION

`main.py`

- **Description**: Defines the Flask application and the `/api/user/<username>` endpoint. Handles GET, POST, and PATCH requests.

```
 """
app.py
------
Flask application for managing user profiles with GET, POST, and PATCH methods.
"""

from flask import Flask, request, jsonify
from utils import get_user_info, list_all_users, create_user_profile, update_user_prof
from constants import (
    METHOD_GET, METHOD_POST, METHOD_PATCH, STATUS_OK, STATUS_CREATED, STATUS_BAD_REQUE
    STATUS_NOT_FOUND, STATUS_FORBIDDEN, LOG_MESSAGES
)
from log import log_message

app = Flask(__name__)

@app.route('/api/user/<username>', methods=[METHOD_GET, METHOD_POST, METHOD_PATCH])
def handle_user(username):
    """
    Handles user profile management.

    Args:
        username (str): The username for the requested operation.

    Returns:
        Response: JSON response with user data or status messages.
    """
    log_message('info', f'Started handling request for username: {username}')

    if request.method == METHOD_GET:
        log_message('info', 'Handling GET request')
        try:
            current_user = request.headers.get('X-Current-User')
            is_admin = request.headers.get('X-Is-Admin') == 'true'
            user_info = get_user_info(username, current_user, is_admin)
            return jsonify(user_info), STATUS_OK
        except ValueError as e:
            log_message('error', str(e))
            return jsonify({"error": str(e)}), STATUS_NOT_FOUND
        except PermissionError as e:
            log_message('error', str(e))
            return jsonify({"error": str(e)}), STATUS_FORBIDDEN

    elif request.method == METHOD_POST:
        log_message('info', 'Handling POST request')
        user_data = request.json
        try:
            created_user = create_user_profile(username, user_data)
            return jsonify(created_user), STATUS_CREATED
        except ValueError as e:
            log_message('error', str(e))
            return jsonify({"error": str(e)}), STATUS_BAD_REQUEST

    elif request.method == METHOD_PATCH:
        log_message('info', 'Handling PATCH request')
        updated_data = request.json
        try:
            current_user = request.headers.get('X-Current-User')
            is_admin = request.headers.get('X-Is-Admin') == 'true'
            if not is_admin and current_user != username:
                raise PermissionError(LOG_MESSAGES['unauthorized_update'].format(curre
            updated_user = update_user_profile(username, updated_data)
            return jsonify(updated_user), STATUS_OK
        except ValueError as e:
            log_message('error', str(e))
            return jsonify({"error": str(e)}), STATUS_BAD_REQUEST
        except PermissionError as e:
            log_message('error', str(e))
            return jsonify({"error": str(e)}), STATUS_FORBIDDEN

    log_message ('info', f'Finished handling request for username: {username}')
    return jsonify({"error": "Invalid method"}), STATUS_BAD_REQUEST

if __name__ == '__main__':
    app.run(debug=True)
```

Functions:

1. **`user_operations(username)`**
    - **Description**: Handles operations based on HTTP method (GET, POST, PATCH) for a specific user.
    - **Parameters**:
        - `username` (str): Username of the user.
    - **Returns**: JSON response with the status of the operation.

LOG.PY

```
"""
log.py
-------
Contains functions for logging messages.
"""

import logging

# Configure logging
logging.basicConfig(filename='app.log', level=logging.INFO, format='%(asctime)s - %(le

def log_message(level, message):
    """
    Logs a message with a specified level.

    Args:
        level (str): The logging level ('info', 'error').
        message (str): The message to log.
    """
    if level == 'info':
        logging.info(message)
    elif level == 'error':
        logging.error(message)
    else:
        logging.warning(f"Unknown log level: {level}. Message: {message}")
```

CONSTANTS.PY

```
"""
constants.py
-------------
Contains constants for the application, including status codes, HTTP methods, and log
"""

VALID_COUNTRY_LIST = ['USA', 'India', 'UK', 'Canada']
EXCLUDED_NUMBERS = ['1234567890', '0987654321']
VALID_GENDERS = ['male', 'female', 'other']
VALID_BLOOD_GROUPS = ['A+', 'A-', 'B+', 'B-', 'AB+', 'AB-', 'O+', 'O-']

# HTTP Status Codes
STATUS_OK = 200
STATUS_CREATED = 201
STATUS_BAD_REQUEST = 400
STATUS_NOT_FOUND = 404
STATUS_FORBIDDEN = 403
```

```
# HTTP Methods
METHOD_GET = 'GET'
METHOD_POST = 'POST'
METHOD_PATCH = 'PATCH'

LOG_MESSAGES = {
    'invalid_email': "Invalid email: {}",
    'valid_email': "Valid email: {}",
    'invalid_age': "Invalid age: {}",
    'valid_age': "Valid age: {}",
    'invalid_mobile': "Invalid mobile number: {}",
    'valid_mobile': "Valid mobile number: {}",
    'invalid_gender': "Invalid gender: {}",
    'valid_gender': "Valid gender: {}",
    'invalid_blood_group': "Invalid blood group: {}",
    'valid_blood_group': "Valid blood group: {}",
    'user_not_found': "User not found: {}",
    'unauthorized_access': "Unauthorized access: {} trying to access {}",
    'user_info': "User info: {}, {}",
    'user_exists': "User already exists: {}",
    'user_created': "User created: {}, {}",
    'user_updated': "User updated: {}, {}",
    'unauthorized_update': "Unauthorized update: {} trying to update {}",
    'unauthorized_list_users': "Unauthorized list users: {}",
    'list_all_users': "Listing all users: {}"
}
```

`utils.py`

- **Description**: Contains utility functions for validating user data and decorators for authentication.

```
"""
utils.py
---------
Contains utility functions for validating user data, retrieving user information, and
"""

import re
from log import log_message
from constants import (
    VALID_COUNTRY_LIST, EXCLUDED_NUMBERS, VALID_GENDERS, VALID_BLOOD_GROUPS, LOG_MESSA
)


def validate_email(email):
    """
    Validates the given email address.

    Args:
        email (str): The email address to validate.

    Raises:
        ValueError: If the email format is invalid.
```

```python
    Returns:
        bool: True if the email is valid, False otherwise.
    """
    log_message('info', f'Started validating email: {email}')

    email_regex = r'^[a-zA-Z0-9_.+-]+@[a-zA-Z0-9-]+\.[a-zA-Z0-9-.]+$'
    if not re.match(email_regex, email):
        log_message('error', LOG_MESSAGES['invalid_email'].format(email))
        raise ValueError(LOG_MESSAGES['invalid_email'].format(email))

    log_message('info', LOG_MESSAGES['valid_email'].format(email))
    log_message('info', f'Finished validating email: {email}')
    return True


def validate_age(age):
    """
    Validates the given age.

    Args:
        age (int): The age to validate.

    Raises:
        ValueError: If the age is invalid.

    Returns:
        bool: True if the age is valid, False otherwise.
    """
    log_message('info', f'Started validating age: {age}')

    if not (0 < age < 120):
        log_message('error', LOG_MESSAGES['invalid_age'].format(age))
        raise ValueError(LOG_MESSAGES['invalid_age'].format(age))

    log_message('info', LOG_MESSAGES['valid_age'].format(age))
    log_message('info', f'Finished validating age: {age}')
    return True


def validate_mobile(mobile):
    """
    Validates the given mobile number.

    Args:
        mobile (str): The mobile number to validate.

    Raises:
        ValueError: If the mobile number is invalid or excluded.

    Returns:
        bool: True if the mobile number is valid, False otherwise.
    """
    log_message('info', f'Started validating mobile number: {mobile}')

    if mobile in EXCLUDED_NUMBERS:
        log_message('error', LOG_MESSAGES['excluded_mobile'].format(mobile))
        raise ValueError(LOG_MESSAGES['excluded_mobile'].format(mobile))

    mobile_regex = r'^\d{10}$'
    if not re.match(mobile_regex, mobile):
        log_message('error', LOG_MESSAGES['invalid_mobile'].format(mobile))
        raise ValueError(LOG_MESSAGES['invalid_mobile'].format(mobile))

    log_message('info', LOG_MESSAGES['valid_mobile'].format(mobile))
    log_message('info', f'Finished validating mobile number: {mobile}')
    return True


def validate_gender(gender):
    """
    Validates the given gender.

    Args:
        gender (str): The gender to validate.

    Raises:
        ValueError: If the gender is invalid.

    Returns:
        bool: True if the gender is valid, False otherwise.
    """
    log_message('info', f'Started validating gender: {gender}')

    if gender not in VALID_GENDERS:
        log_message('error', LOG_MESSAGES['invalid_gender'].format(gender))
        raise ValueError(LOG_MESSAGES['invalid_gender'].format(gender))

    log_message('info', LOG_MESSAGES['valid_gender'].format(gender))
    log_message('info', f'Finished validating gender: {gender}')
    return True


def validate_blood_group(blood_group):
    """
    Validates the given blood group.

    Args:
        blood_group (str): The blood group to validate.

    Raises:
        ValueError: If the blood group is invalid.

    Returns:
        bool: True if the blood group is valid, False otherwise.
    """
    log_message('info', f'Started validating blood group: {blood_group}')

    if blood_group not in VALID_BLOOD_GROUPS:
        log_message('error', LOG_MESSAGES['invalid_blood_group'].format(blood_group))
        raise ValueError(LOG_MESSAGES['invalid_blood_group'].format(blood_group))
    log_message('info', LOG_MESSAGES['valid_blood_group'].format(blood_group))
    log_message('info', f'Finished validating blood group: {blood_group}')
    return True


def validate_user_data(func):
    """
    Decorator function to validate user data (email, age, mobile, gender, blood group)

    Args:
        func (function): The function to wrap.

    Returns:
        function: The wrapped function.
    """

    def wrapper(*args, **kwargs):
        log_message('info', 'Started validating user data')
        user_data = kwargs.get('user_data', {})
        email = user_data.get('email')
        age = user_data.get('age')
        mobile = user_data.get('mobile')
        gender = user_data.get('gender')
        blood_group = user_data.get('blood_group')

        if email:
            validate_email(email)
        if age:
            validate_age(age)
        if mobile:
            validate_mobile(mobile)
        if gender:
            validate_gender(gender)
        if blood_group:
            validate_blood_group(blood_group)

        result = func(*args, **kwargs)
        log_message('info', 'Finished validating user data')
        return result

    return wrapper


@validate_user_data
def get_user_info(username, current_user, is_admin, user_data=None):
    """
    Retrieves information for the specified user.

    Args:
        username (str): The username of the user whose information is to be retrieved.
        current_user (str): The username of the current user making the request.
        is_admin (bool): Whether the current user is an admin.

    Raises:
        PermissionError: If the current user is not authorized to view the user's info
        ValueError: If the user is not found.

    Returns:
        dict: The user's information.
    """
    log_message('info', f'Started retrieving user info for: {username}')

    from data import data

    if username not in data["records"]:
        log_message('error', LOG_MESSAGES['user_not_found'].format(username))
        raise ValueError(LOG_MESSAGES['user_not_found'].format(username))

    if not is_admin and current_user != username:
        log_message('error', LOG_MESSAGES['unauthorized_access'].format(current_user,
        raise PermissionError(LOG_MESSAGES['unauthorized_access'].format(current_user,

    user_info = data["records"][username]
    log_message('info', LOG_MESSAGES['user_info'].format(username, user_info))
    log_message('info', f'Finished retrieving user info for: {username}')
    return user_info


def list_all_users(current_user, is_admin):
    """
    Lists all users.

    Args:
        current_user (str): The username of the current user making the request.
        is_admin (bool): Whether the current user is an admin.

    Raises:
        PermissionError: If the current user is not authorized to list all users.

    Returns:
        dict: A dictionary of all users.
    """
    log_message('info', f'Started listing all users by: {current_user}')

    from data import data

    if not is_admin:
        log_message('error', LOG_MESSAGES['unauthorized_list_users'].format(current_us
        raise PermissionError(LOG_MESSAGES['unauthorized_list_users'].format(current_u

    all_users = data["records"]
    log_message('info', LOG_MESSAGES['list_all_users'].format(current_user))
    log_message('info', f'Finished listing all users by: {current_user}')
    return all_users


def create_user_profile(username, user_data):
    """
    Creates a new user profile.
    Args:
        username (str): The username of the new user.
        user_data (dict): The data for the new user.

    Raises:
        ValueError: If the user already exists or the data is invalid.

    Returns:
        dict: The newly created user profile.
    """
    log_message('info', f'Started creating user profile for: {username}')

    from data import data

    if username in data["records"]:
        log_message('error', LOG_MESSAGES['user_exists'].format(username))
        raise ValueError(LOG_MESSAGES['user_exists'].format(username))

    user_data['role'] = 'user'
    data["records"][username] = user_data
    log_message('info', LOG_MESSAGES['user_created'].format(username, user_data))
    log_message('info', f'Finished creating user profile for: {username}')
    return user_data


def update_user_profile(username, updated_data):
    """
    Updates an existing user profile.

    Args:
        username (str): The username of the user to update.
        updated_data (dict): The updated data for the user.

    Raises:
        ValueError: If the user is not found or the data is invalid.

    Returns:
        dict: The updated user profile.
    """
    log_message('info', f'Started updating user profile for: {username}')

    from data import data

    if username not in data["records"]:
        log_message('error', LOG_MESSAGES['user_not_found'].format(username))
        raise ValueError(LOG_MESSAGES['user_not_found'].format(username))

    data["records"][username].update(updated_data)
    log_message('info', LOG_MESSAGES['user_updated'].format(username, updated_data))
    log_message('info', f'Finished updating user profile for: {username}')
    return data["records"][username]
```

Functions:

1. **`validate_email(email)`**
    - **Description**: Validates email format.
    - **Parameters**:
        - `email` (str): Email address.
    - **Returns**: Boolean indicating if the email is valid.
1. **`validate_age(age)`**
    - **Description**: Validates age.
    - **Parameters**:
        - `age` (int): Age.
    - **Returns**: Boolean indicating if the age is valid.
1. **`validate_mobile(mobile)`**
    - **Description**: Validates mobile number.
    - **Parameters**:
        - `mobile` (str): Mobile number.
    - **Returns**: Boolean indicating if the mobile number is valid.
1. **`validate_gender(gender)`**
    - **Description**: Validates gender.
    - **Parameters**:
        - `gender` (str): Gender.
    - **Returns**: Boolean indicating if the gender is valid.
1. **`validate_blood_group(blood_group)`**
    - **Description**: Validates blood group.
    - **Parameters**:
        - `blood_group` (str): Blood group.
    - **Returns**: Boolean indicating if the blood group is valid.
1. **`validate_user_data(func)`**
    - **Description**: Decorator to validate user data.
    - **Parameters**:
        - `func` (function): Function to wrap.
    - **Returns**: Wrapped function.
1. **`get_user_info(username, current_user, is_admin, user_data=None)`**
    - **Description**: Retrieves user information.
    - **Parameters**:
        - `username` (str): Username of the user.
        - `current_user` (str): Requester's username.
        - `is_admin` (bool): Indicates if the requester is an admin.

        - `user_data` (dict): Optional user data.
    - **Returns**: User information as a dictionary.

1. **`create_user_profile(username, user_data)`**

    - **Description**: Creates a new user profile.
    - **Parameters**:
        - `username` (str): Username of the new user.
        - `user_data` (dict): User data.
    - **Returns**: Created user profile.

1. **`update_user_profile(username, user_data, current_user, is_admin)`**

    - **Description**: Updates an existing user profile.
    - **Parameters**:
        - `username` (str): Username of the user.
        - `user_data` (dict): User data to update.
        - `current_user` (str): Requester's username.
        - `is_admin` (bool): Indicates if the requester is an admin.
    - **Returns**: Updated user profile.

`data.py`

- **Description**: Contains user data for the application.

```
"""
data.py
---------
Contains user data for the project.
"""

data = {
    "records": {
        "kiran": {"email": "kiran@example.com", "age": 25, "mobile": "9876543210", "ge
        "radha": {"email": "radha@example.com", "age": 30, "mobile": "9123456789", "ge
        "nkiran": {"email": "nkiran@example.com", "age": 22, "mobile": "9988776655", '
    }
}
```

Data Structure:

1. **`data`**:

    - **Type**: Dictionary
    - **Structure**:

```
{
  "records": {
    "username": {
      "email": "string",
      "age": int,
      "mobile": "string",
      "gender": "string",
      "blood_group": "string",
      "role": "string"
    }
  }
}
```

---

log.py

```python
"""
log.py
-------
Contains functions for logging messages.
"""

import logging

# Configure logging
logging.basicConfig(filename='app.log', level=logging.INFO, format='%(asctime)s - %(le

def log_message(level, message):
    """
    Logs a message with a specified level.

    Args:
        level (str): The logging level ('info', 'error').
        message (str): The message to log.
    """
    if level == 'info':
        logging.info(message)
    elif level == 'error':
        logging.error(message)
    else:
        logging.warning(f"Unknown log level: {level}. Message: {message}")
```

4. LOG FILE INFORMATION

- **Location**: `app.log`
- **Log Messages**: Include information on function starts, ends, validation results, and errors.

5. TESTING

USE POSTMAN TO TEST THE ENDPOINTS:

- **GET**: Retrieve user information.
- **POST**: Create a new user profile.
- **PATCH**: Update an existing user profile.

Ensure the request headers and bodies match the API specifications.

Testing applications

POST Create User Profile:

- **Method**: POST

PATCH Update User Profile:

- **Method**: PATCH