UserManagementSystem Documentation

1. INTRODUCTION

Project Overview

- This project aims to create a User Management System in Python.
- It includes functionalities for user validation, data management, and role-based access control.

Key Objectives

- Validate user data (email, age, mobile number, gender, blood group).
- Implement role-based access control (admin and normal user).
- Provide CRUD operations for user management.

2. PROJECT STRUCTURE

Directory Structure

```
bash
Copy code
/project
│   app.log        # Log file for recording application activities
│   constants.py   # Defines constants and configurations
│   log.py         # Logging setup and utilities
│   utils.py       # Utility functions for validation and user management
│   main.py        # Main application script demonstrating user scenarios
│   data.py        # Data storage module for user records
```

3. Component Explanation

3.1 `constants.py`

Purpose

- Defines constants used throughout the project.

Constants

- `VALID_COUNTRY_LIST`: Valid country codes for mobile number validation.
- `EXCLUDED_NUMBERS`: List of excluded mobile numbers.
- `GENDER_OPTIONS`: Valid options for gender.
- `BLOOD_GROUP_OPTIONS`: Valid options for blood groups.
- `LOG_SWITCH`: Toggle for logging functionality.

3.2 `log.py`

Purpose

- Configures logging for the application.

Functionality

- Provides functions for logging messages at different levels (`debug`, `info`, `warning`, `error`, `critical`).
- Logs messages to `app.log` based on configured logging level and settings in `constants.py`.

3.3 `utils.py`

Purpose

- Contains utility functions for user data validation and management.

Functions

- `validate_email(email)`: Validates email addresses.
- `validate_age(age)`: Validates age within a specified range.
- `validate_mobile(mobile)`: Validates mobile numbers and checks against excluded numbers.
- `validate_gender(gender)`: Validates gender against predefined options.
- `validate_blood_group(blood_group)`: Validates blood groups against predefined options.
- Additional functions for user management operations (e.g., add user, update user).

3.4 `data.py`

Purpose

- Stores and manages user data.

Data Structure

- Uses a dictionary (`data['records']`) to store user information, including username, email, age, mobile number, gender, blood group, and role.

3.5 `main.py`

Purpose

- Demonstrates various user scenarios using the implemented functionalities.

Scenarios

- **Admin Viewing Specific User**: Fetches detailed information about a specific user.
- **Admin Listing All Users**: Lists all users and their details.
- **Normal User Viewing Own Information**: Displays detailed information for the logged-in normal user.

---

4. RUNNING THE PROJECT

Execution

- Run `main.py` to execute the project.
- Example command:

```
python main.py
```

---

5. LOGGING AND ERROR HANDLING

Log File

- `app.log` stores all application activities and errors.
- Located in the project directory for easy access and review.

Logging Levels

- Utilizes different logging levels (`debug`, `info`, `warning`, `error`, `critical`) to categorize and prioritize log messages.
- Controlled by `LOG_SWITCH` in `constants.py`.

---

6. CONCLUSION

Summary

- This User Management System project showcases essential functionalities for validating user data and managing user records.
- It ensures role-based access control and provides comprehensive logging for monitoring application activities.

Code

CONSTANTS.PY

```python
"""
constants.py
-------------
Contains constants used throughout the project.
"""

# List of valid country codes
VALID_COUNTRY_LIST = ["91", "45", "67", "56"]

# List of excluded mobile numbers
EXCLUDED_NUMBERS = ["9898989898", "9999999999", "8888888888"]

# List of valid genders
VALID_GENDERS = ["male", "female", "other"]
```

```python
# List of valid blood groups
VALID_BLOOD_GROUPS = ["A+", "A-", "B+", "B-", "O+", "O-", "AB+", "AB-"]

# Log switch (True to enable logging, False to disable)
LOG_SWITCH = True
```

LOG.PY

```python
"""
log.py
-------
Sets up logging for the project.
"""

import logging
from constants import LOG_SWITCH

# Create a custom logger
logger = logging.getLogger(__name__)
logger.setLevel(logging.DEBUG)

# Create handlers
file_handler = logging.FileHandler('app.log')
console_handler = logging.StreamHandler()

# Set level of handlers
file_handler.setLevel(logging.DEBUG)
console_handler.setLevel(logging.DEBUG)

# Create formatters and add it to handlers
formatter = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s - %(message)s')
file_handler.setFormatter(formatter)
console_handler.setFormatter(formatter)

# Add handlers to the logger
logger.addHandler(file_handler)
logger.addHandler(console_handler)


def log_message(level, message):
    """
    Logs a message with the given log level if logging is enabled.

    Args:
        level (str): The level of the log (e.g., 'debug', 'info', 'warning', 'error',
        message (str): The message to log.
    """
    if LOG_SWITCH:
        if level == 'debug':
            logger.debug(message)
        elif level == 'info':
            logger.info(message)
        elif level == 'warning':
            logger.warning(message)
        elif level == 'error':
            logger.error(message)
        elif level == 'critical':
            logger.critical(message)
```

DATA.PY

```
"""
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", '

    }
}
```

UTILS.PY

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

import re
from log import log_message
from constants import VALID_COUNTRY_LIST, EXCLUDED_NUMBERS, VALID_GENDERS, VALID_BLOOD


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.

    Returns:
        bool: True if the email is valid, False otherwise.
    """
    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', f"Invalid email format: {email}")
        raise ValueError("Invalid email format")
    log_message('info', f"Valid 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 not within the valid range (0-120).

    Returns:
        bool: True if the age is valid, False otherwise.
    """
    if not (0 <= age <= 120):
        log_message('error', f"Invalid age: {age}")
        raise ValueError("Invalid age")
    log_message('info', f"Valid 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 format is invalid.

    Returns:
        bool: True if the mobile number is valid, False otherwise.
    """
    mobile_regex = r'^\d{10}$'
    if not re.match(mobile_regex, mobile):
        log_message('error', f"Invalid mobile number: {mobile}")
        raise ValueError("Invalid mobile number")
    if mobile in EXCLUDED_NUMBERS:
        log_message('info', f"Excluded mobile number: {mobile}")
        return False
    log_message('info', f"Valid 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 not valid.

    Returns:
        bool: True if the gender is valid, False otherwise.
    """
    if gender.lower() not in VALID_GENDERS:
        log_message('error', f"Invalid gender: {gender}")
        raise ValueError("Invalid gender")
    log_message('info', f"Valid 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 not valid.

    Returns:
        bool: True if the blood group is valid, False otherwise.
    """
    if blood_group.upper() not in VALID_BLOOD_GROUPS:
        log_message('error', f"Invalid blood group: {blood_group}")
        raise ValueError("Invalid blood group")
    log_message('info', f"Valid blood group: {blood_group}")
    return True


def get_user_info(username, current_user, is_admin):
    """
    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 requested u
        ValueError: If the requested user is not found.

    Returns:
        dict: The user information if the user is found and the current user is author
    """
    from data import data
    user_info = data['records'].get(username)
    if user_info:
        if username == current_user or is_admin:
            log_message('info', f"User info for {username}: {user_info}")
            return user_info
        else:
            log_message('warning', f"Unauthorized access attempt by {current_user} to
            raise PermissionError("Unauthorized access")
    else:
        log_message('error', f"User {username} not found")
        raise ValueError("User not found")


def list_all_users(current_user, is_admin):
    """
    Lists all users if the requester is an admin.

    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 containing all users' information.
    """
    from data import data
    if is_admin:
        log_message('info', f"Admin {current_user} listing all users")
        return data['records']
    else:
        log_message('warning', f"Unauthorized access attempt by {current_user} to list
        raise PermissionError("Unauthorized access")


def add_user(username, email, age, mobile, gender, blood_group, role, current_user, is
    """
    Adds a new user to the system.

    Args:
        username (str): The username of the new user.
        email (str): The email address of the new user.
        age (int): The age of the new user.
        mobile (str): The mobile number of the new user.
        gender (str): The gender of the new user.
        blood_group (str): The blood group of the new user.
        role (str): The role of the new user (admin or user).
        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 add users.
        ValueError: If any of the user details are invalid.

    Returns:
        dict: The updated records with the new user added.
    """
    from data import data
    if is_admin:
        validate_email(email)
        validate_age(age)
        validate_mobile(mobile)
        validate_gender(gender)
        validate_blood_group(blood_group)
        if username in data['records']:
            log_message('error', f"User {username} already exists")
            raise ValueError("User already exists")
        data['records'][username] = {
            "email": email,
            "age": age,
            "mobile": mobile,
            "gender": gender,
            "blood_group": blood_group,
            "role": role
        }
        log_message('info', f"Admin {current_user} added new user {username}")
        return data['records']
    else:
        log_message('warning', f"Unauthorized access attempt by {current_user} to add
        raise PermissionError("Unauthorized access")


def update_user(username, updates, current_user, is_admin):
    """
    Updates an existing user's information.

    Args:
        username (str): The username of the user to update.
        updates (dict): A dictionary of the updates to apply.
        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 update users.
        ValueError: If any of the updated user details are invalid.

    Returns:
        dict: The updated user information.
    """
    from data import data
    user_info = data['records'].get(username)
    if not user_info:
        log_message('error', f"User {username} not found")
        raise ValueError("User not found")

    if current_user != username and not is_admin:
        log_message('warning', f"Unauthorized access attempt by {current_user} to upda
        raise PermissionError("Unauthorized access")

    if 'email' in updates:
        validate_email(updates['email'])
    if 'age' in updates:
        validate_age(updates['age'])
    if 'mobile' in updates:
        validate_mobile(updates['mobile'])
    if 'gender' in updates:
        validate_gender(updates['gender'])
    if 'blood_group' in updates:
        validate_blood_group(updates['blood_group'])

    data['records'][username].update(updates)
    log_message('info', f"User {current_user} updated user {username}: {updates}")
    return data['records'][username]
```

MAIN.PY

```
"""
main.py
--------
Demonstrates various user scenarios including admin and normal user actions.
"""

from utils import (
    validate_email, validate_age, validate_mobile, validate_gender, validate_blood_gro
    get_user_info, list_all_users, add_user, update_user
)
from log import log_message


def main():
    """
    Main function demonstrating various user scenarios.
    """
    # Scenarios

    # Admin adding a new user
    try:
        admin_username = "kiran"
        new_user_data = {
            "username": "dummy",
            "email": "dummy@example.com",
            "age": 30,
            "mobile": "7776543210",
            "gender": "male",
            "blood_group": "A+",
            "role": "user"
        }
        updated_records = add_user(
            new_user_data['username'],
            new_user_data['email'],
            new_user_data['age'],
            new_user_data['mobile'],
            new_user_data['gender'],
            new_user_data['blood_group'],
            new_user_data['role'],
            admin_username,
            is_admin=True
        )
        log_message('info', f"Admin {admin_username} added new user {new_user_data['us
    except (ValueError, PermissionError) as e:
        log_message('critical', str(e))

    # Admin updating a user
    try:
        admin_username = "kiran"
        user_to_update = "dummy"
        updates = {"email": "new_dummy@example.com"}
        updated_user_info = update_user(user_to_update, updates, admin_username, is_ad
        log_message('info', f"Admin {admin_username} updated user {user_to_update}: {u
    except (ValueError, PermissionError) as e:
        log_message('critical', str(e))

    # Admin viewing a specific user information
    try:
        admin_username = "kiran"
        user_to_view = "ndines"
        user_info = get_user_info(user_to_view, admin_username, is_admin=True)
        log_message('info', f"Admin {admin_username} viewed user {user_to_view}: {user
    except (ValueError, PermissionError) as e:
        log_message('critical', str(e))

    # Admin listing all users
    try:
        admin_username = "nkiran"
        all_users = list_all_users(admin_username, is_admin=True)
        log_message('info', f"Admin {admin_username} listed all users: {all_users}")
    except (ValueError, PermissionError) as e:
        log_message('critical', str(e))

    # Normal user viewing their own information
    try:
        normal_username = "radha2"
        user_info = get_user_info(normal_username, normal_username, is_admin=False)
        log_message('info', f"Normal user {normal_username} viewed their information:
    except (ValueError, PermissionError) as e:
        log_message('critical', str(e))


# Run the main function
main()
```