Educational Management System Documentation

Name : Hosam Mohammed Ahmed Yousif

Github link :

Introduction

This document provides comprehensive documentation for the educational management system code, covering functionalities, class structures, relationships, and usage examples.

System Overview

This system is designed to manage users, courses, and user interactions within an educational setting. Here's a breakdown of the key functionalities:

- **User Management:**
  - Create user accounts with different roles (e.g., Admin, Doctor/Teacher, Teaching Assistant, Student).
  - Securely store passwords using hashing algorithms (bcrypt).
  - Implement role-based access control to restrict functionalities based on user roles.
- **Course Management:**
  - Create courses with details like name, code, start/end dates, and unregister deadlines.

    - Allow authorized users (Doctors/Teachers) to create and edit courses.
- **Student Management:**
    - Allow students to register for courses within the registration period.
    - Enable students to unregister from courses before the deadline.
    - Provide students with functionalities to view their registered courses and assignments.
    - Allow students to submit solutions for assignments.

User Interface (UI) Considerations

While this code focuses on the backend functionalities, creating a user interface (UI) is essential for interacting with the system. The UI could be a web application, desktop application, or a mobile app, depending on the specific needs. The UI should provide functionalities to:

- Manage user accounts (login, registration, profile management).
- Manage courses (create, edit, view).

- Allow students to register/unregister for courses.
- Enable students to view assignments and submit solutions.
- Provide role-specific dashboards for different user types (Admin, Doctor/Teacher, Student).

Code Structure

The code is organized into several modules/files:

- **user_classes.py:** Contains class definitions for User, AdminUser, Doctor (Teacher), TeachingAssistant, and Student.
- **login.py:** Implements functionalities for user login and authentication.
- **interface.py:** Defines functions for displaying menus based on user roles.
- **education_db.py:** Provides functions for connecting to the database, managing database interaction (potentially includes helper functions for specific database operations).
- **connect.py:** (Might be a separate file) Handles database connection logic using the odbc module (for MS SQL Server in this example).

- **Courses.py:** Defines the Course class with functionalities for creating, retrieving, and editing courses.

Detailed Class Documentation

1. User Class

Python

class User:

```python
class User( RoleMixin):
    def __init__(self, user_id, username, password, email,role):
        self.user_id = user_id
        self.username = username
        self.password = password
        self.email = email
        self.role = role
```

The User class serves as the base class for all user types in the system. It defines attributes common to all users:

- user_id: Unique identifier for the user (assumed to be retrieved from the database).
- username: Username used for login.

- password: Hashed password for secure storage (bcrypt recommended).
- email: User's email address.
- role: User's role (e.g., "admin", "doctor", "teaching_assistant", "student").

2. RoleMixin Class

```python
class RoleMixin:

    @classmethod
    def role_check(cls, allowed_roles):
        def decorator(func):
            def wrapper(self, *args, **kwargs):
                if not any(isinstance(self, role) for role in allowed_roles):
                    raise PermissionError("Unauthorized access for this role")
                return func(self, *args, **kwargs)
            return wrapper
        return decorator
```

The RoleMixin class provides a decorator-based approach for implementing role-based access control (RBAC). Here's how it works:

- The role_check class method is a decorator factory that takes a list of allowed roles.

- The decorator created by role_check wraps the decorated function and checks if the current user instance (self) has any of the allowed roles.
- If the user role is not authorized for the function, a PermissionError is raised.

3. AdminUser Class (inherits User)

```
from User import User
from RoleMixin import  RoleMixin
class AdminUser(User):
    @RoleMixin.role_check(["admin"])
    def access_admin_panel(self):
        print("Accessing admin panel...")
```

The AdminUser class inherits from User and represents an admin user with specific functionalities. The

The AdminUser class example demonstrates the usage of the RoleMixin decorator. The access_admin_panel function is decorated with @RoleMixin.role_check(["admin"]). This ensures that only users with the "admin" role can access this functionality.

4. Doctor/Teacher Class (inherits User)

```
class Doctor(User):
    def __init__(self, user_id, username, password, email,role, courses=[]):
        super().__init__(user_id, username, password, email)
        self.courses = courses

    @RoleMixin.role_check("doctor")
```

The Doctor class inherits from User and represents a doctor (or teacher) user with functionalities related to managing courses. Similar to AdminUser, methods like create_course and edit_course are decorated with @RoleMixin.role_check(["doctor"]) to restrict access to authorized users.

5. TeachingAssistant Class (inherits User)

```
class TeachingAssistant(User):
    def __init__(self, user_id, username, password, email,role, courses=[]):
        super().__init__(user_id, username, password, email)
        self.courses = courses
```

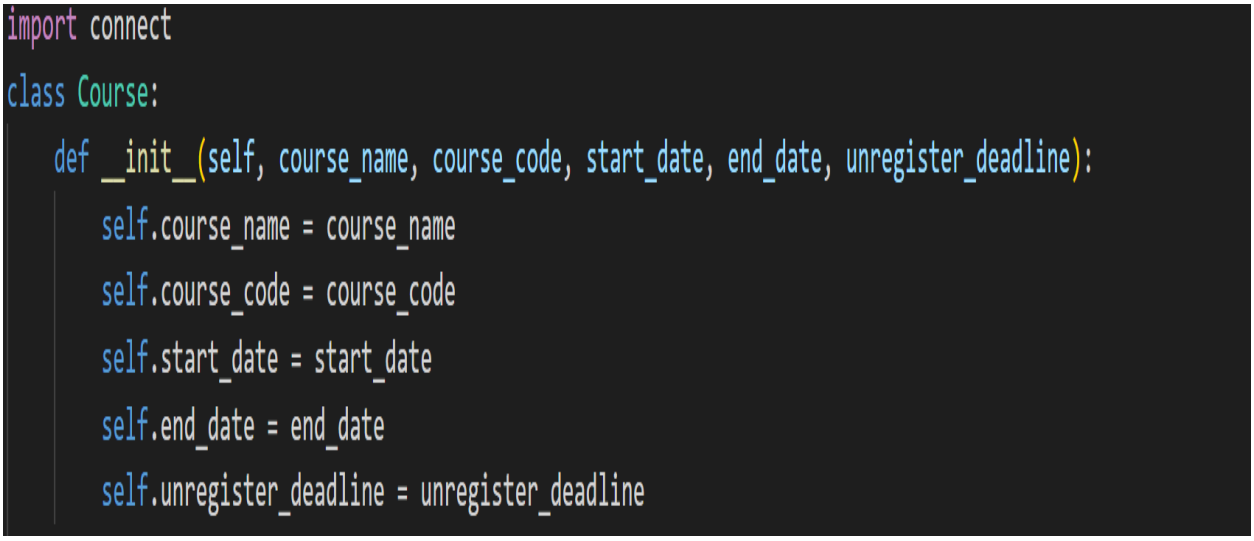
The TeachingAssistant class inherits from User and represents a teaching assistant with potentially limited functionalities compared to a Doctor/Teacher. The view_course_details method allows teaching assistants to view existing courses but might not grant permission to create or edit them (depending on system requirements).

6. Student Class (inherits User)

```python
class Student(User):
    def __init__(self, user_id, username, password, email,role,courses=[]):
        super().__init__(user_id, username, password, email)
        self.courses = courses

    @RoleMixin.role_check("student")
    def register_for_course(self, course):

        if not isinstance(course, Course):
            raise ValueError("Invalid course provided. Please provide a Course object.")

        existing_course = Course.get_course(course.course_code)
        if not existing_course:
            raise ValueError(f"Course with code '{course.course_code}' not found.")

        current_date = datetime.date.today()
        if current_date < existing_course.registration_start or current_date > existing_course.registration_end:
            raise ValueError(f"Registration for '{existing_course.course_name}' (code: {existing_course.course_code}) is closed.")
        else:
            print(f"You have successfully registered for '{existing_course.course_name}' (code: {existing_course.course_code}).")
```

The Student class inherits from User and represents a student user with functionalities for course registration, viewing assignments, and submitting solutions.

7. Course Class



```
import connect
class Course:
    def __init__(self, course_name, course_code, start_date, end_date, unregister_deadline):
        self.course_name = course_name
        self.course_code = course_code
        self.start_date = start_date
        self.end_date = end_date
        self.unregister_deadline = unregister_deadline
```

The Course class represents a course offered in the system. It defines attributes like course name, code, start/end dates, and the deadline for students to unregister.

Database Interaction (education_db.py)

This section outlines the functionalities provided in the education_db.py module for interacting with the database:

1. **connect_to_database():** Establishes a connection to the database using the credentials and connection string.
2. **execute_query(query, params=None):** Executes a provided SQL query with optional parameters and returns the results.

Educational Management System Documentation (Continued)

Database Interaction (education_db.py) (Continued)

3. **fetch_all(cursor):** Fetches all rows from the database cursor object and returns them as a list.
4. **fetch_one(cursor):** Fetches a single row from the database cursor object and returns it as a tuple.
5. **close_connection():** Closes the connection to the database.

```python
import pypyodbc as odbc
driver_name='SQL SERVER'
server = 'DESKTOP-ALUKKGH\SQLEXPRESS'
database = 'education_system'

connection_string = f"""
DRIVER={{{driver_name}}};
SERVER={server};
DATABASE={database};
Trust Connection=yes;"""
def get_connection():
    try:
        connection = odbc.connect(connection_string)
        print("Connected to database successfully!")
        return connection
    except :
        print("Error connecting to database:")
        return None

get_connection()
```

Error Handling

The code should implement proper error handling mechanisms using exceptions. Here are some examples:

- **ValueError:** Raised for invalid data types passed to functions (e.g., an integer expected for course_id but a string is provided).
- **PermissionError:** Raised by the RoleMixin decorator when a user attempts to access unauthorized functionalities.
- **DatabaseError:** Raised for database connection errors or query execution issues.

Logging

Consider implementing a logging mechanism to record system events, errors, and user actions. This can be helpful for debugging, auditing, and analyzing system usage.

Security Considerations

- **Password Hashing:** Always store passwords securely using a hashing algorithm like bcrypt. Never store plain text passwords in the database.

- **Input Validation:** Validate user input to prevent SQL injection attacks and other security vulnerabilities.
- **Regular Updates:** Keep the system software libraries and database up-to-date to address known security issues.

Deployment Considerations

- **Web Application:** If deploying the system as a web application, choose a secure web server framework and follow best practices for web application security.
- **Desktop Application:** For desktop applications, implement access control mechanisms and consider code signing to ensure application integrity.

Further Enhancements

- **Assignment Management:** This documentation focuses on basic functionalities. You can extend the system to include features for managing assignments, deadlines, and grading.
- **Communication Features:** Implement functionalities for sending notifications or

announcements to users regarding courses, assignments, or system updates.

- **Reporting:** Develop reports for admins to analyze student enrollment, course performance, and other relevant data.
- **User Interface:** As mentioned earlier, this documentation focuses on the backend. Design a user-friendly UI (web, desktop, or mobile) for users to interact with the system effectively.

Conclusion

This comprehensive documentation provides a detailed explanation of the Educational Management System's code structure, functionalities, class interactions, and database interaction. Remember to adapt and extend this documentation based on the specific implementation details and functionalities of your code.