Module 5 Assignment: Flask Backend Basics

Objective

To build a structured backend for the IELTS Speaking Test platform using Flask. This assignment focuses on project structuring, database integration with SQLAlchemy, and creating RESTful APIs for managing user and test data.

Scenario

The IELTS Speaking Test platform needs a robust backend system to store and retrieve data for users and test sessions. As a backend developer, you are tasked with organizing the Flask project structure, integrating a MySQL database, and implementing APIs to handle user registration and test data. The backend will serve as the foundation for the platform's functionality.

Requirements

1. **Flask Project Structure**:
    - Organize the backend codebase into the following files:
        - **`app.py`**: Entry point of the application.
        - **`models.py`**: Define SQLAlchemy models for database tables.
        - **`config.py`**: Configure database connection and application settings.
    - Use Flask blueprints to modularize routes (e.g., separate routes for users and tests).
2. **Database Integration with SQLAlchemy**:
    - Configure SQLAlchemy to connect to a MySQL database.
    - Define the following models:
        - **User**: Fields: `id` (primary key), `name`, `email`, `phone`, `created_at`.
        - **SpeakingTest**: Fields: `id` (primary key), `user_id` (foreign key), `test_date`, `status`, `score`.
        - **ListeningTest**: Fields: `id` (primary key), `user_id` (foreign key), `test_date`, `status`, `score`.
    - Use Flask-Migrate to handle database migrations.
3. **RESTful API Design**:
    - Implement the following API endpoints:
        - **`POST /api/users`**: Register a new user. Accept user details in the request body and store them in the database. Validate the input (e.g., email format, phone number length).
        - **`GET /api/users`**: Retrieve all registered users. Support optional query parameters for pagination (`page`, `limit`).
        - **`GET /api/users/<id>`**: Retrieve details of a specific user by their ID.

        - **`POST /api/speaking-tests`**: Schedule a new speaking test for a user. Accept `user_id`, `test_date`, and `status` in the request body.
        - **`GET /api/speaking-tests/<id>`**: Retrieve details of a specific speaking test.
4. **Middleware for Validation and Error Handling**:
    - Add middleware to validate API inputs (e.g., check for missing fields, invalid data types).
    - Handle errors gracefully with appropriate status codes and error messages.

Deliverables

1. Flask project folder with the following:
    - **`app.py`**: Main application file.
    - **`models.py`**: Database models.
    - **`config.py`**: Application and database configuration.
    - **Routes**: Modularized route files for users and tests.
    - **Database migrations**: Flask-Migrate files for creating and updating tables.
2. Postman collection or cURL commands to test the implemented API endpoints.
3. A README file describing:
    - How to set up and run the Flask application.
    - The database schema and API specifications.
    - Examples of valid requests and responses for each endpoint.

Submission Guidelines

1. Submit the project folder in a zipped file named `Module5_Assignment_<YourName>.zip`.
2. Include the database migration files for SQLAlchemy.
3. Ensure all routes, models, and configurations are well-commented and follow best practices.
4. Provide screenshots or logs demonstrating successful API calls and database interactions.

Evaluation Criteria

1. **Project Structure and Organization (20%)**:
    - Proper organization of files and modularization of routes.
    - Clean and readable code with appropriate comments.
2. **Database Integration (25%)**:
    - Correct setup and configuration of SQLAlchemy with MySQL.
    - Accurate definition of models and relationships.
3. **API Functionality (30%)**:
    - APIs should work as specified and handle edge cases (e.g., invalid user ID, missing fields).

- Pagination and query parameters should function correctly.

4. **Middleware and Error Handling (15%)**:
    - Validation logic should correctly handle invalid inputs.
    - Errors should return meaningful status codes and messages.
5. **Documentation (10%)**:
    - README should include clear setup instructions, API specs, and testing examples.