## Module 7 Assignment: Backend Advanced Features

---

### Objective

To enhance the Flask backend for the IELTS Speaking Test platform by implementing advanced features such as asynchronous programming, Azure OpenAI integration for generating questions, and robust middleware and logging systems.

---

### Scenario

As the IELTS Speaking Test platform scales, advanced backend functionalities are required to handle real-time requests, generate AI-based IELTS questions, and ensure secure and efficient operation. In this task, you will implement asynchronous APIs, integrate Azure OpenAI, and configure middleware and logging for better performance and traceability.

---

### Requirements

1. **Asynchronous Programming in Flask**:
    - Update the Flask application to support asynchronous routes using `asyncio`.
    - Implement an asynchronous API endpoint:
        - **`GET /api/questions`**: Fetch IELTS speaking test questions from a mock database (or generate placeholders).
        - Ensure the route handles real-time requests efficiently with minimal delay.
2. **Azure OpenAI Integration**:
    - Connect the backend to Azure OpenAI for generating IELTS speaking questions.
    - Steps:
        - Set up the Azure OpenAI API and securely store the API key in a `.env` file.
        - Create an API endpoint:
            - **`POST /api/generate-question`**: Accept a `topic` in the request body and return a generated question based on that topic.
            - Example Request:

```
{
  "topic": "Education"
}
```

            - Example Response:

```
{
  "question": "What are the advantages of online
learning in today's world?"
```

```
}
```

        - Handle errors such as invalid API keys or rate limits gracefully.
3. **Middleware for Validation and Logging**:
    - Add middleware to validate request payloads:
        - Ensure `topic` is present in the body for the `POST /api/generate-question` endpoint.
        - Return a clear error response if validation fails.
    - Configure logging for:
        - API usage: Log each request with the endpoint, timestamp, and response status.
        - Error tracking: Log errors with details such as request data and exception traceback.
    - Store logs in a file (`logs/api.log`) for persistent access.

---

**Deliverables**

1. Flask project folder containing:
    - **`app.py`**: Main application file with asynchronous routes.
    - **`config.py`**: Configuration for environment variables and logging setup.
    - **`routes/question_routes.py`**: Routes for question generation and fetching.
    - **`middleware.py`**: Middleware for validation and logging.
    - **`.env`**: Environment variables (e.g., Azure OpenAI API key).
2. Postman collection or cURL commands to demonstrate:
    - `GET /api/questions`: Fetching placeholder or mock questions.
    - `POST /api/generate-question`: Generating a question using Azure OpenAI.
3. A README file with:
    - Setup instructions for running the application.
    - Steps for integrating Azure OpenAI and testing the APIs.
    - Examples of log outputs for successful and failed requests.

---

**Submission Guidelines**

1. Submit the project in a zipped folder named `Module7_Assignment_<YourName>.zip`.
2. Include a sample `.env` file (without the actual API key) for reference.
3. Provide sample log files (`logs/api.log`) showcasing both successful requests and error handling.
4. Ensure all code is well-commented and follows Flask best practices.

---

**Evaluation Criteria**

1. **Asynchronous Programming (30%)**:
    - Proper implementation of asynchronous APIs using `asyncio`.

    - Efficient handling of real-time requests with minimal latency.
2. **Azure OpenAI Integration (30%)**:
    - Correct connection to the Azure OpenAI API and secure API key handling.
    - The question generation API should work as specified, handling errors gracefully.
3. **Middleware and Logging (25%)**:
    - Validation middleware should correctly handle invalid requests.
    - Logging should capture detailed API usage and error information.
4. **Documentation and Logs (15%)**:
    - README should clearly explain the setup and usage of features.
    - Logs should be well-structured and include relevant details.