API

Part 1: Backend API Development "Library Management API"

Objective

Create a small RESTful API for managing a collection of books in a library. The API should handle several operations related to books and authors. A database (SQL or NoSQL) can be used but is not mandatory; you can store the data in a JSON file (e.g., an array or object).

Problem Statement

You are tasked with building a dockerized API for a Library Management System that allows users to perform the following operations:

1. Add a new book
   Each book should have the following details:
   - Title (required)
   - Author (required)
   - Published Year (required)
   - ISBN (required)
   - Genre (optional)
2. List all books
   Return a list of all books with their details.
3. Search for books
   Allow filtering books by one or more of the following:
   - Author
   - Published Year
   - Genre
4. Delete a book by ISBN
   Remove a specific book from the library using its ISBN.
5. Update book details by ISBN
   Update one or more details of a specific book using its ISBN.

Instructions

1. Implementation

- Choose any backend language and framework (e.g., Node.js, Python, Java, etc.).
- The API must follow RESTful principles.
- Store data in memory, DB or JSON file (database isn't mandatory).

2. Routes and Functionality
   The following are the functionalities the API must support. The choice of HTTP methods, request parameters, and responses is up to you.

Feature	**Description**
Add a new book	Add a new book with the specified details.
List all books	Return a list of all books in the library.
Search for books	Filter books by author, published year, or genre.
Delete a book by ISBN	Remove a specific book from the library using its unique ISBN.
Update book details	Update details of an existing book using its unique ISBN.

3. API Documentation (Swagger) :
   - The API documentation will be created using **Swagger**.
   - The Swagger UI should be accessible in the API after running it, allowing for an interactive view of the API documentation.
   - Documentation will be included in the project and can be accessed via a route like `/api-docs`.

---

4. Deliverables:

- **Backend Project Repository**:
  i. A **GitHub repository** containing the complete **dockerized backend code**.
  ii. The repository should include the following files and directories:
    1. **Backend source code**: API code implementing the Library Management System containing **The Swagger UI should be accessible via a specific endpoint, like `/api-docs`.**
    2. **`Dockerfile`**: Instructions for building the Docker image for the backend.
    3. **`README.md`**: A detailed README file with instructions for:

a. **Building and running the Docker container**: Step-by-step instructions to build and run the Dockerized API.
b. **Accessing the Swagger API Documentation**: Instructions on how to access the Swagger documentation (e.g., at `/api-docs`), where users can interactively test API endpoints.

4. A **Postman collection** containing requests for all API functionalities.

5. Additional Notes
    - Your API should return responses in JSON format.

Evaluation Criteria

- Code quality and readability.
- Correctness of the implementation.
- RESTful standards (use of proper HTTP methods and status codes).
- Completeness of the Postman collection.
- Swagger Integration.
- Docker Integration.

CI/CD

Part 2: Branch Strategy and CI/CD using GitHub Actions Workflows for Library Management API Project

Objectives

Establish a clear branch strategy for separating development, production, and deployment tasks, while implementing GitHub Actions workflows to automate code validation, testing, and deployment processes. This will ensure high code quality, error-free builds, and seamless deployment to Azure, automating the entire pipeline to minimize manual intervention and enable faster, more reliable updates to the live environment.

Branch Strategy

1. **Main Branch**:
    - **Purpose**: Serves as the stable branch containing production-ready code.
    - **Usage**: All changes must be tested and approved before merging into this branch to ensure the stability and reliability of the Library Management API.
2. **Backend Branch**:
    - **Purpose**: Focuses on the development of the backend API for managing library books and authors.
    - **Usage**: Isolates backend-related changes, preventing any disruptions to the main branch. API-specific features, such as adding books, updating book details, or searching books, will be developed here.
3. **Deploy Branch**:
    - **Purpose**: Dedicated to deployment-related activities.
    - **Usage**: This branch contains the code that will be deployed to Azure. Any changes pushed to this branch will trigger the deployment pipeline, ensuring only validated code from the main branch is deployed.

GitHub Actions Workflows

1. **Pre-Merge Workflow**:
    - **Trigger**: Runs on pull request creation or updates targeting the main branch.
    - **Purpose**: Validates the new code before it is merged into the main branch, ensuring high-quality standards.

    - **Outcome**: Ensures that only thoroughly tested, linted, and error-free code is merged into the stable branch, reducing the risk of bugs in the production system.
2. **Component Build Workflow**:
    - **Trigger**: Runs on pushes to the **backend** branch.
    - **Purpose**: Focuses on testing and building only the backend-related changes.
        - **Backend Tests**: Ensures that API functionalities work correctly.
        - **Backend Build**: Compiles and builds the backend components, preparing the code for the production environment.
    - **Outcome**: Provides rapid feedback on the backend code, allowing backend developers to address issues quickly without triggering unnecessary workflows for non-backend changes.
3. **Deployment Workflow**:
    - **Trigger**: Runs when changes are pushed to the **deploy** branch.
    - **Purpose**: Automates the deployment of the Library Management API to **Azure**.
        - **Automated Deployment**: Pushes the latest, tested version of the code from the **main branch** to the Azure environment.
        - **CI/CD Pipeline**: Reduces manual deployment tasks, ensuring a faster, more reliable deployment process for the Library Management System API.
    - **Outcome**: Ensures that only validated, high-quality code from the **main branch** is deployed to production, minimizing errors in the live environment.

Deliverables

Include the following in your LibraryAPI project GitHub repository:

1. **GitHub Actions Workflow Configuration**:
    - **Pre-Merge Workflow**:
        - YAML file located in `.github/workflows/pre-merge.yml` for validating pull requests targeting the main branch.
        - Includes unit tests, lint checks, and basic code analysis to maintain code quality for the Library Management API.
    - **Component Build Workflow**:
        - YAML file located in `.github/workflows/component-build.yml` to test and build changes pushed to the **backend** branch.
        - Focuses on backend-specific testing and builds to ensure API functionality and performance.

    - **Deployment Workflow**:
        - YAML file located in `.github/workflows/deployment.yml` to deploy code pushed to the **deploy** branch to Azure.
        - Automates the deployment process for seamless integration with the Azure cloud environment.
2. **CI/CD Pipeline Implementation**:
    - Fully implemented CI/CD pipeline, ensuring automated tests, builds, and deployments to **Azure** for the Library Management System API.
    - Provides end-to-end automation from development to production.
3. **Deployment Environment Setup**:
    - Properly configured **Azure deployment environment**, including any necessary configuration for the CI/CD pipeline, ensuring a seamless deployment and integration process for the Library Management API.

Evaluation Criteria:

- **Branch Strategy:** Proper separation of branches (main, backend, deploy).
- **GitHub Actions Workflow:** Correct workflow triggers and configurations.
- **CI/CD Pipeline:** Full automation of code validation, testing, building, and deployment.
- **Deployment:** Seamless deployment to Azure.
- **Documentation:** Clear instructions and well-documented YAML files for workflows.

DP_SOLID

Part 3 : Design Patterns in a Pizza Restaurant

Objective:

The goal of this assignment is to assess your ability to apply design patterns to solve real-world problems. You will design and implement a pizza restaurant ordering system using multiple design patterns.

Scenario

You are tasked with building a simple system for a pizza restaurant. The restaurant has two types of pizzas: Margherita and Pepperoni. Customers can customize their pizzas with optional toppings such as Cheese, Olives, and Mushrooms.

Each pizza object should (at least) have methods to show description and to calculate cost

Check the provided code in *starter.py* file

Requirements

1. Creating the pizza is considered a difficult process. An object should be responsible for creating the base pizzas(`Margherita` or `Pepperoni`).
2. Allow customers to add 1 or more toppings dynamically.
    - Toppings should include:
        - Cheese (Cost: $1)
        - Olives (Cost: $0.5)
        - Mushrooms (Cost: $0.7)
    - The name of the topping should be appended to the description and the price should be added to the cost.
3. Complete the Implementation of an Inventory Manager to manage ingredient availability. Only one inventory manager should be allowed in the system.
4. Make at least 2 payment methods (e.g. Paypal, Credit Card).
5. Output:
    - Display the final description of the pizza (base type + toppings) and the total cost.
    - Confirm the payment method used (e.g., PayPal or Credit Card) and display a success message upon payment completion.

Deliverables

- A Python script that demonstrates the following:
    1. Create a Margherita pizza.
    2. Add Cheese and Olives as toppings.
    3. Calculate and print the total cost and description.
    4. Implement a payment method to simulate the payment process after calculating the total cost.

Design patterns

For each design pattern you applied to your system, you need to:

1. Describe the pattern and its application to your system.
2. Explain how the system was designed before applying the pattern.
3. Illustrate how the pattern improves the maintainability and extensibility of the project.
4. Provide code snippets or examples where applicable, showing how the design pattern was integrated into the project.

Discuss the concept of Over Engineering and provide a code snippet example related to the Pizza restaurant project.

Describe both in the DesignPatterns.md `file.`

Relation between SOLID principles and Design patterns:

For each design pattern applied in your pizza restaurant ordering system, describe which SOLID principles the pattern addresses or aligns with. Explain how the pattern helps implement these principles in your code, in the SOLID_DP.md file.

Deliverables:

Push your pizza restaurant code into a GitHub repository.

Include two files in this repository:

1. **SOLID_DP.md** file where you:
    - Describe the design patterns applied in your code as mentioned above.
    - Explain how each design pattern adheres to specific SOLID principles.

2. **DesignPatterns.md** file where you:
    - Provide an explanation for each design pattern as mentioned above and the concept of overengineering, including code examples to illustrate how overengineering may occur in your system.

Provide a link to the GitHub repository containing your code, **SOLID_DP.md**, and **DesignPatterns.md** files.

Evaluation Criteria:

1. Code Quality and Readability
2. Correctness of Implementation
3. Design Patterns Application
4. SOLID Principles Implementation
5. Overengineering Concept
6. Extensibility and Maintainability

Rubric

Rubric for the Assignment (2.5%)

Part 1: Library Management API (0.9%)

- **Functionality (0.3%)**: Correct implementation of CRUD operations and proper use of HTTP methods.
- **Swagger Documentation (0.2%)**: Functional Swagger UI with clear, interactive API docs.
- **Dockerization (0.2%)**: Dockerized API with correct `Dockerfile` and setup instructions.
- **Code Quality (0.2%)**: Clean, modular code with proper error handling and validation.

Part 2: Branch Strategy & CI/CD (0.5%)

- **CI/CD Pipeline (0.5%)**: Proper GitHub Actions workflows for testing, building, and deploying to Azure.

Part 3: Pizza Restaurant Design Patterns (1.0%)

- **Design Patterns (0.3%)**: Correct application of patterns.
- **SOLID Principles (0.3%)**: Clear explanation of SOLID principles in `SOLID_DP.md`.
- **Overengineering (0.2%)**: Explanation of overengineering with code examples.
- **Code Quality (0.2%)**: Modular, maintainable code with proper payment and pizza creation logic.

Additional Criteria

- **Documentation & Readability (0.1%)**: Well-written README, clear instructions, and clean code comments.

Total: 2.5%

Will only be added to the labs and quizzes grades, not the midterm or project.

Best of Luck !