**Lab Report: Bazar.com - Multi-tier Online Book Store**

**Introduction:**

The provided code implements a microservices-based online book store named Bazar.com. The system consists of three main components: a front-end server and two back-end servers (catalog server and order server). The microservices communicate through HTTP REST calls, and the implementation utilizes the Flask web framework in Python.

**Code Structure:**

The codebase is well-organized, with distinct files such as **app.py** and **book.py**. This modular structure enhances maintainability and readability.

**Database Configuration:**

SQLite is chosen as the database backend, providing a lightweight solution for storing book information persistently. The SQLAlchemy library is used to interact with the database, and the Marshmallow library aids in object serialization.

**Database Initialization:**

The **database\_init** list is used to populate the database with initial book entries upon its creation. This ensures that the system starts with predefined book data.

**Book Class:**

The **Book** class is defined to model the book entity. It inherits from SQLAlchemy's **Model** class and includes fields such as **id**, **title**, **topic**, **quantity**, and **price**. The class provides methods for searching, retrieving, and updating book information.

**Code Explanation:**

* The **search** method performs case-insensitive queries based on the book's topic.
* The **get** method retrieves a book by its ID.
* The **update** method allows updating book fields, considering optional parameters.

**Marshmallow Schema:**

Marshmallow schemas (**TopicSchema**, **ItemSchema**, **UpdateSchema**) are employed for formatting responses in JSON. These schemas facilitate consistent and structured data presentation.

**Code Explanation:**

* Schemas define the expected structure of JSON responses for different endpoints.
* The **jsonify** method is used to convert objects into JSON format.

**Microservices Endpoints:**

1. **Home Endpoint:**
   * The **/** endpoint returns a simple greeting message.
2. **Search Endpoint:**
   * The **/search/<method>/<param>** endpoint handles search queries based on either item or topic. It dynamically selects the appropriate handler based on the specified method.

**Code Explanation:**

* The **/search** endpoint dynamically chooses the search method based on the provided parameter.
* The **queries** dictionary centralizes search methods, making it easy to extend functionality.

1. **Modify Endpoint:**
   * The **/modify/<book\_id>** endpoint allows updating book information, such as quantity and price.

**Code Explanation:**

* The **/modify** endpoint uses the **update** method in the **Book** class to modify book details.
* Input parameters are taken from the JSON payload in the request.

1. **Buy Endpoint:**
   * The **/buy/<book\_id>** endpoint initiates the purchase process, interacting with the catalog server to decrement book quantities.

**Code Explanation:**

* The **/buy** endpoint queries the catalog server for book availability before completing a purchase.
* It utilizes the **requests** library to communicate with the catalog server.

**Communication Between Microservices:**

* The front-end server communicates with the catalog server to search for books based on topic or item.
* The order server communicates with the catalog server to check book availability before completing a purchase.
* The front-end server connects with both the order and catalog services through HTTP REST calls.

**Code Explanation:**

* The **requests** library is used for seamless communication between microservices.
* The front-end server acts as a central point, orchestrating interactions with both the order and catalog services.

**Error Handling:**

* The code handles various error scenarios gracefully, returning informative messages and appropriate HTTP status codes.

**Code Explanation:**

* Error handling is implemented for cases such as invalid book IDs, unsuccessful searches, or failed updates.
* The code returns meaningful error messages along with the corresponding HTTP status codes.

**Microservices Enhancement:**

1. **Concurrency:**
   * The code leverages Flask's built-in support for handling multiple concurrent requests. The use of threads or async processing is managed by the Flask framework.
2. **Persistence:**
   * While the code currently uses SQLite, a lightweight database, for simplicity, future improvements could involve switching to a more scalable database solution for larger-scale applications.
3. **Communication Optimization:**
   * The code efficiently interacts with external microservices through HTTP REST calls. However, further optimization could involve implementing asynchronous communication for improved responsiveness.

**Future Improvements:**

* Enhance error handling and provide more detailed error messages.
* Implement authentication and authorization mechanisms for secure transactions.
* Consider implementing logging for better traceability and debugging.

**Instructions for Running the Program:**

1. Clone the repository from GitHub.
2. Install the required dependencies using **pip install -r requirements.txt**.
3. Run the Flask application using **python app.py**.
4. Access the endpoints through a web browser or a tool like **curl**.

**Output:**

* Provide a sample output log demonstrating the execution of different functionalities, such as book searches and purchases.

**Test API :**

