PHENIKAA UNIVERSITY

PHENIKAA SCHOOL OF COMPUTING

Layered Architecture Implementation – CRUD

Course: Software Architecture

Course Class: CSE703110-1-2-25(N02)

Group 7:

1. Le Thi Kieu Trang **ID: 23010502**

2. Quach Huu Nam **ID: 23012358**

3. Trieu Tien Quynh **ID: 23010648**

Hanoi, January 7, 2026

Lab 3 – Layered Architecture Implementation - CRUD

1. Abstract

The objective of this lab is to implement a Layered Architecture to ensure the separation of concerns between the Presentation, Business Logic, and Data Persistence layers.

+ Lab Requirements: Build CRUD operations (Create, Read, Update, Delete) for the Product entity, strictly adhering to the data flow.

+ Actual Implementation: The team applied this architecture to the Student Management Microservice (Student Service). Instead of using an in-memory list, the team utilizes MySQL for persistent storage.

2. Architectural Design

Based on the source code in the src/microservices/student_service directory, the system is divided into the following layers:

Layer	Component	Source File	Primary Responsibility
Presentation Layer	Controller / API	app.py	Handles HTTP Requests (GET, POST, PUT, DELETE), validates input data, and returns JSON responses.
Data Model	Entity	models.py	Defines the data structure of the Student object for data exchange between layers.
Persistence Layer	Repository	repository.py	Executes SQL statements to interact directly with the MySQL database.

Note: In this implementation, simple Business Logic is integrated directly into the Controller (app.py) to reduce complexity for the microservice.

3. Implementation Details

- ❖ Data Model (Entity)

The Student class defined in models.py acts as a Data Transfer Object (DTO). It contains attributes such as student_id, student_name, student_email and a to_dict() method to support serialization to JSON.

Code Illustration (models.py)

```python
class Student:
    def __init__(self, student_id, student_name, date_of_birth, gender, student_class, student_email, cohort=None):
        self.student_id = student_id
        self.student_name = student_name
        self.date_of_birth = date_of_birth
        self.gender = gender
        self.student_class = student_class
        self.student_email = student_email
        self.cohort = cohort

    def to_dict(self):
        return {
            'student_id': self.student_id,
            'student_name': self.student_name,
            'date_of_birth': self.date_of_birth,
            'gender': self.gender,
            'student_class': self.student_class,
            'student_email': self.student_email,
            'cohort': self.cohort
        }
```

❖ Persistence Layer (Repository)

The StudentRepository class in repository.py is responsible for opening connections to MySQL and executing SQL statements. The CRUD methods are implemented as follows:

+ Create: Uses INSERT INTO

+ Read: Uses SELECT

+ Update: Uses UPDATE

+ Delete: Uses DELETE

Code Illustration (repository.py):

```python
class StudentRepository:
    def __init__(self): ...
        }

    def _get_conn(self):
        return mysql.connector.connect(**self.db_config)

    def add_student(self, student): ...
            conn.close()

    def get_student_by_id(self, s_id): ...
            conn.close()

    def get_all_students(self): ...
            conn.close()

    def update_student(self, student):
        conn = self._get_conn()
        cursor = conn.cursor()
        try:
            sql = "UPDATE students SET student_name = %s, date_of_birth = %s, gender = %s, student_class = %s, student_email = %s, cohort = %s WHERE student_id = %s"
            cursor.execute(sql, (student.student_name, student.date_of_birth, student.gender, student.student_class, student.student_email, student.cohort, student.student_id))
            conn.commit()
            return cursor.rowcount > 0
        finally:
            cursor.close()
            conn.close()
```

❖ Presentation Layer (Controller)

The app.py file uses Flask to define API Endpoints. It acts as the Controller, receiving requests from the client, calling the Repository to process data, and returning the results.

Code Illustration (app.py):

```
    repo = StudentRepository()

    # ==========================================
    # 1. API TẠO SINH VIÊN (POST) & LẤY LIST (GET)
    # ==========================================
    @app.route("/api/students", methods=["POST", "GET"])
> def handle_students(): ...
                return jsonify({"error": str(ex)}), 400

    # ==========================================
    # 2. API LẤY THÔNG TIN SINH VIÊN (GET), UPDATE (PUT) & XÓA (DELETE)
    # ==========================================
    @app.route("/api/students/<student_id>", methods=["GET", "PUT", "DELETE"])
> def student_detail(student_id): ...
                return jsonify({"error": str(ex)}), 500

    if __name__ == "__main__":
        # Chạy Port 5001
        app.run(debug=True, port=5001)
```

4. Testing

The system has completed the following APIs, adhering to RESTful standards:

+ POST /api/students: Creates a new student. Data is validated at the Controller level before being persisted to the DB.

+ GET /api/students: Retrieves a list of all students.

+ GET /api/students/<id>: Retrieves detailed information of a specific student. Returns 404 if not found.

+ PUT /api/students/<id>: Updates student information.

+ DELETE /api/students/<id>: Removes a student from the system.

5. Conclusion

Lab 3 has been completed through the construction of the Student Service.

+ Architecture: Achieved clear separation between API logic handling code (app.py) and data access code (repository.py).

+ Data: Successfully utilized the Student Model to ensure data consistency.

+ Scalability/Extension: Upgraded from the requirement of using a List (in-memory/temporary storage) to using MySQL (persistent database), better meeting the requirements for subsequent Labs.