PHENIKAA UNIVERSITY

PHENIKAA SCHOOL OF COMPUTING

ACADEMIC MANAGEMENT SYSTEM

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, February 6, 2026

TASK ASSIGNMENT TABLE

No.	Student ID	Full Name	Assigned Tasks
1	23010502	Le Thi Kieu Trang (Leader)	Part 2 – Requirements Analysis Core Functional Requirements Use Case Modeling: + System Overview Use Case Diagram + Administrator Use Cases Part 3 – Architecture Design System Context and Scope Architectural Pattern: Microservices Architecture Client-Side Architecture Part 4 – Testing Testing Strategy (Unit Testing & Integration Testing) Part 5 – Documentation Conclusion Final Documentation Review

PREFACE

In the context of rapid digital transformation, the application of information technology in higher education management is no longer just an option but an inevitable trend. Traditional academic management systems, or legacy Monolithic systems, often face significant challenges regarding scalability, maintenance, and the integration of new features as the volume of students and data continues to grow.

Recognizing these challenges, our team decided to undertake the project titled "Building an Academic Management System based on Microservices Architecture." The project focuses on addressing the management of student information, courses, credit enrollment, grading, and more, by decomposing the system into independent services that communicate flexibly via standard RESTful APIs.

This report details the system analysis, design, and implementation process. It covers the construction of core services such as Identity, Student, and Grade Services using Python (Flask), as well as the design of a highly interactive Single Page Application (SPA) user interface. Notably, the project delves into specific distributed architecture techniques, such as Asynchronous Communication and shared database management.

We hope that the results of this project will serve as a useful reference model for the application of modern software architecture to real-world management problems.

To complete this project, alongside the efforts of our team members, we have received dedicated attention, encouragement, and guidance from M.S.Vu Quang Dung. Throughout the implementation of this topic, he spared no effort in imparting knowledge, guiding our problem-solving mindset, and providing valuable feedback to help us refine the system, ranging from Microservices organization to source code optimization.

Although the team has made every effort to apply learned knowledge to practice, due to time constraints and limited practical experience, the project inevitably contains shortcomings. We sincerely look forward to receiving your feedback to further improve the topic and gain valuable lessons for our future careers.

We would like to express our sincere gratitude!

1. Overview

The project focuses on the design and implementation of an "Academic Management System" aimed at supporting the credit-based training model. The system addresses the management requirements for university, faculty, major, course, and student information; facilitates students in course registration and tracking academic results. Simultaneously, it allows lecturers to enter and manage grades, ensuring transparency and accuracy in the academic assessment process.

Throughout the development process, the system has successfully transitioned from a Monolithic architecture to a Microservices Architecture.

The final system comprises:

+ Independent Microservices: Identity, Student, Course, Grade, Enrollment, Email, Faculty, University, and Major Service (including KKT Service).
+ Modern Frontend: Built using Vanilla JavaScript (Single Page Application), enabling smooth interaction for both Students and Lecturers without page reloads.
+ Centralized Database with Logical Separation: The system currently uses a shared MySQL database schema, where each microservice accesses only the tables relevant to its own business domain. This design ensures clear logical separation of data ownership while simplifying deployment and development. The architecture is designed to be transition-ready toward a full Database-per-Service model in future iterations.
+ Hybrid Communication: Combines REST API (Synchronous) and Threading (Asynchronous).

The achieved result is a system with high availability, capable of handling large traffic volumes during peak course registration periods.

2. Project Requirements & Goals

2.1 Core Functional Requirements

The system fulfills the following key functional requirements:

ID	Function	Detailed Description
FR-01	Catalog Management	Administrator can add, edit, or delete Student/ University/ Faculty / Major and Course information (CRUD).
FR-02	Course Registration	Students can log in and self-register for courses opened in the current semester. The system validates course eligibility and records the enrollment.

FR-03	Grading & GPA Calculation	Lecturers enter component scores (Attendance, Mid-term, Final) with customizable weights. The system automatically calculates the overall score (10-point scale) and converts it to letter grades (A, B, C...) and the 4.0 scale score immediately upon entry.
FR-04	Asynchronous Notification	The system automatically sends notification emails to students immediately after the Lecturer completes grading (background processing).
FR-05	Result Lookup	Students can view detailed transcripts of completed courses and their Cumulative GPA.
FR-06	Security & Authorization	The system requires authentication and enforces access authorization (Administrator/Faculty/Student) via the Identity Service.

2.4 Use case modeling

- ❖ Use Case Diagram

Figure 1: Use Case Diagram – User Interaction

This diagram presents the primary interactions between external actors and the system, focusing on architecturally significant use cases rather than detailed operational behaviors.

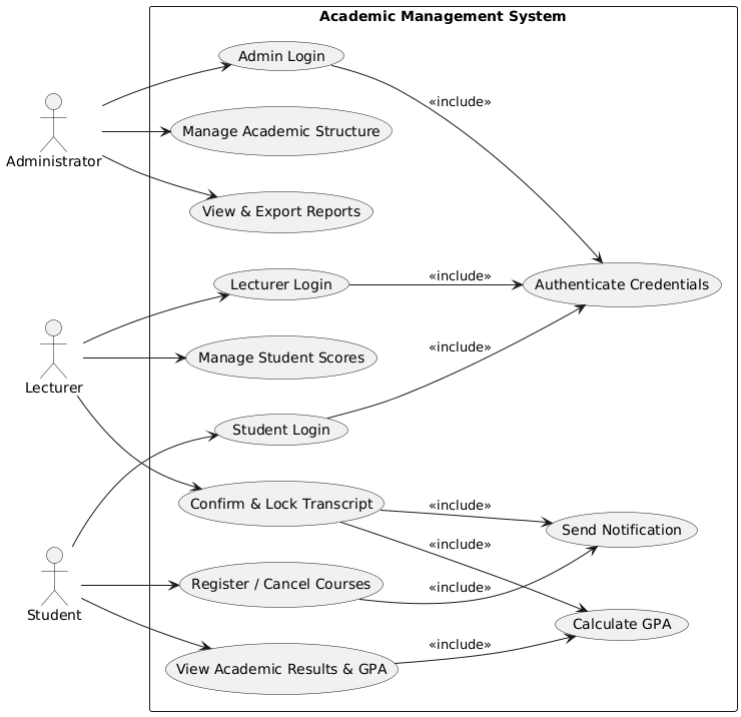
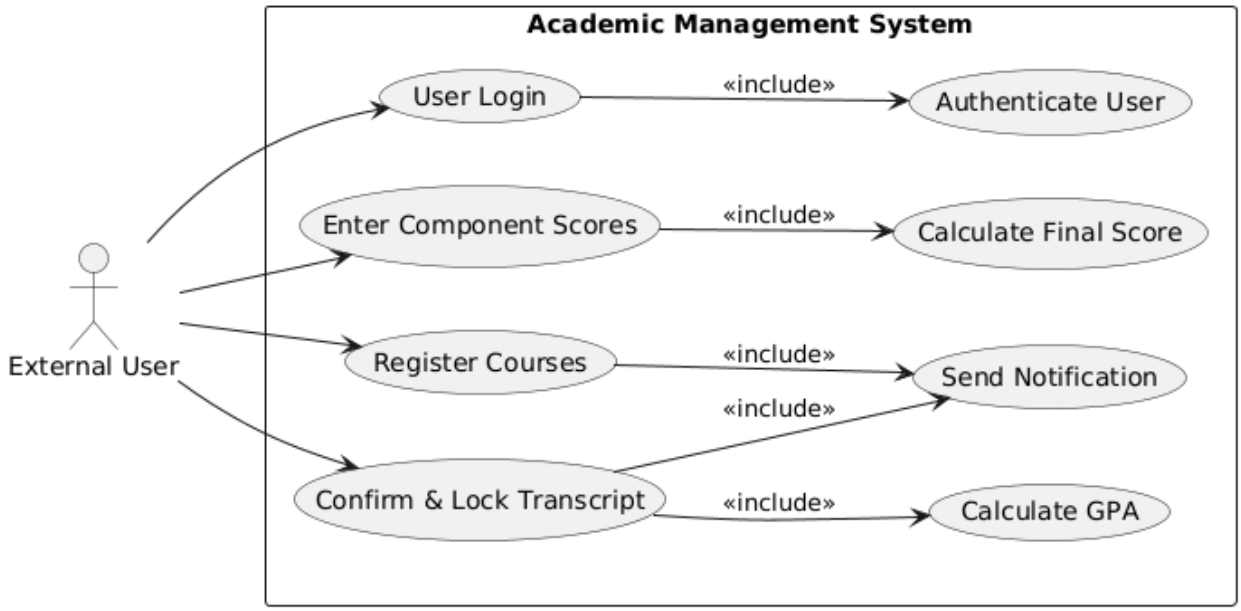


Figure 2: Use Case Diagram – System Functionalities

In the Use Case Diagram – System Functionalities, the system itself is not modeled as an actor. Instead, internal system behaviors such as authentication, GPA calculation, and notification sending are represented as included use cases using the <<include>> relationship, as they are automatically executed as part of user-initiated actions.

The User actor represents a generalized external user, while role-specific interactions are detailed in other diagrams.



- ❖ Specification and Construction of Administrator Use Case Group

Overview Description

The Administrator is the actor responsible for managing core academic and organizational data within the system. Administrator interactions mainly focus on system access control and high-level data management, including universities, faculties, majors, courses, and student records. In addition, the Administrator is authorized to monitor academic outcomes by viewing and exporting confirmed grade sheets and generating summary reports to support management and assessment activities.

Use Case Specification

+ UC-A1 Administrator Login

Use Case Name	Administrator Login		
Created By	Development Team	Last Updated By	System Officer
Created Date	Jan 1, 2026	Last Modified Date	Jan 8, 2026
Description	Allows the Administrator to authenticate their identity using a username and password		

	to access the system's administrative functions. This function utilizes the centralized Identity Service.
Actors	- Administrator (Primary Actor) - Identity Service (Authentication System)
Pre-conditions	1. The Identity Service is operational. 2. The user has accessed the Login page. 3. The Administrator account exists in the system database.
Post-conditions	Success: 1. The system returns an Access Token (e.g., fake-jwt-token-for-Administrator). 2. The user is redirected to the Administrator Dashboard. 3. The Administratoristration menu displays full functions according to permissions. Failure: 1. The user remains on the login page. 2. The system displays a corresponding error message.
Main Flow	1. Access Page: - Administrator accesses the admin portal URL; the system displays the Login Form. 2. Enter Credentials: - Administrator enters Username and Password. - Clicks the "Login" button. 3. Validation: - The System (Frontend) performs preliminary validation: Fields must not be empty. 4. Send Request: - Frontend sends an HTTP POST request to the API. - Payload: { "username": "admin", "password": "..." } 5. Authentication (Backend): - Identity Service receives the request. - Checks if the username exists. - Compares the submitted password with the stored hash in the Database. 6. Generate Token: - If credentials are correct, Identity Service generates a success response. - Response: { "message": "Login successful", "token": "...", "role": "admin" } 7. Redirect: - Frontend receives the Token and saves it (LocalStorage/Cookie).

	- Frontend checks the role: If role == "admin" → Redirect to the Dashboard. 8. Use Case Ends.
Alternative Flow	Step 5a. Incorrect credentials: → Identity Service does not find the user or the password matches. → Returns HTTP 401 with JSON: { "error": "Invalid credentials" }. → Frontend displays a red notification: "Incorrect username or password." Step 7a. Non-Administrator privileges: → User logs in with a valid account but the role is "Student" or "Lecturer". → System detects incorrect role permission. → Notification: "You do not have permission to access the Administrator page" and redirects to the appropriate portal.
Exceptions	1. Service Connection Loss: → Identity Service (Port 5004) is down. → Frontend receives a connection error (Connection Refused). → Displays notification: "System is under maintenance, please try again later."
Requirements	1. Security: Passwords sent must be encrypted via HTTPS (in Production environment). 2. Performance: Login response time must be < 1 second.

+ UC-A2 Manage Academic Structure

Use Case Name	Manage Academic Structure		
Created By	Development Team	Created By	Development Team
Created Date	Jan 1, 2026	Created Date	Jan 1, 2026
Description	Allows Administrators to manage the hierarchical organization of the institution. This includes managing Universities (Root), Faculties (Child of University), and Majors (Child of Faculty).		
Actors	- Administrator (Primary Actor) - Uni Service (Manages Universities) - Faculty Service (Manages Faculties) - Major Service (Manages Majors)		
Pre-conditions	1. Administrator has successfully logged in. 2. All structure-related services (uni, faculty, major) are operational.		
Post-	Success:		

conditions	1. Structure data is created/updated/deleted in the respective Databases. 2. The hierarchical tree is refreshed on the interface. Failure: 1. Data remains unchanged. 2. Error message displayed (e.g., Constraint Violation).
Main Flow	1. Select Entity Type: - Administrator selects "Academic Structure" menu. - Chooses the entity to manage: University, Faculty, or Major. 2. View List (Read): - System calls the corresponding API based on selection: + University: GET /api/universities + Faculty: GET /api/faculties (requires University ID filter) + Major: GET /api/majors (requires Faculty ID filter) - Displays the data table. 3. Add New Entity (Create): - Admin clicks "Add New". - If University: Enters Code, Name, Address. - If Faculty: Selects Parent University → Enters Code, Name - If Major: Selects Parent Faculty → Enters Code, Name. - Clicks "Save". - System calls POST to the respective Service. 4. Update Entity (Update): - Admin selects a row and modifies information. - Clicks "Update". - System calls PUT API to save changes. 5. Delete Entity (Delete): - Admin clicks "Delete". - System checks for child data (Referential Integrity). - If safe, System calls DELETE API. - System notifies: "Deleted successfully". 6. Use Case Ends.
Alternative Flow	Step 3a. Duplicate Code: → Administrator enters a code that already exists. → Service returns 409 Conflict.

	→ System notifies: "Entity Code already exists." Step 5a. Data Integrity Violation (Delete Blocked): → Administrator tries to delete a Faculty that still has Majors assigned. → System blocks the request. → Notification: "Cannot delete this Faculty because it contains Majors. Please remove child data first."
Exceptions	1. Service Unavailable: → One of the microservices is down. → System displays: "Unable to load structure data. Please try again later."
Requirements	1. Hierarchy Rule: A Major must belong to a Faculty; a Faculty must belong to a University. 2. Uniqueness: Codes must be unique within their scope (e.g., Major Codes must be unique within a Faculty).

+ UC-A3 View & Export Reports

Use Case Name	View & Export Reports		
Created By	Development Team	Created By	Development Team
Created Date	Jan 1, 2026	Created Date	Jan 1, 2026
Description	Allows Administrators to view academic statistics (e.g., Enrollment status, Grade distribution, Student list per major) and export these reports to external files (Excel) for administrative purposes.		
Actors	- Administrator (Primary Actor) - Grade Service (Data Source) - Enrollment Service (Data Source) - Student Service (Data Source)		
Pre-conditions	1. Administrator has successfully logged into the system. 2. Data exists in the system (Students, Grades, Enrollments). 3. Relevant services are operational.		
Post-conditions	Success: 1. Statistical data is displayed visually (Table/Chart). 2. The report file is successfully downloaded to the Administrator's device. Failure: 1. Error notification is displayed.		

	2. File download fails.
Main Flow	1. Access Report Dashboard: - Administrator selects the "Reports & Statistics" menu. - System displays a list of available report types (e.g., "Semester GPA Summary", "Enrollment Statistics", "Student List"). 2. Configure Parameters: - Administrator selects a Report Type. - Sets filters: Semester (e.g., Fall 2025), Faculty (e.g., IT), Major. - Clicks "Generate Report". 3. Data Retrieval: - System calls the relevant API (e.g., GET /api/grades/reports/gpa or GET /api/enrollment/stats). - The Service aggregates data and returns JSON. 4. View Report: - System renders the data into a Table or Chart on the screen. 5. Export Data: - Administrator clicks "Export to Excel" (or PDF). - System calls the Export API (e.g., GET /api/grades/reports/export?format=xlsx). - The Backend generates the file and streams it back. - The Browser initiates the file download. 6. Use Case Ends.
Alternative Flow	Step 3a. No Data Found: → The filter criteria yield no results (e.g., A new semester with no grades yet). → System notifies: "No data available for the selected criteria." → Export button is disabled. Step 5a. Large Dataset (Async Export): → The report contains thousands of records (taking > 30s to generate) → System notifies: "Report is being generated. You will be notified when it is ready." → The task is pushed to a Background Queue.
Exceptions	1. Service Timeout: → The aggregation query takes too long, causing a Gateway Timeout (504). → System suggests: "Data is too large. Please narrow down the date range or use Export feature. 2. File Generation Error:

	→ Error occurs during file creation (e.g., Library failure). → Notification: "Failed to generate file. Please contact IT support."
Requirements	1. Formats: Must support .xlsx (Excel) for data manipulation. 2. Accuracy: Statistics must reflect real-time data (or cached data not older than 24 hours, depending on configuration).

3. Architectural Design & Implementation

3.1 System Context and Scope

Actor:

Student: The primary end-user of the system.

- Role: Accesses the system to view personal profiles and academic curricula, register for credit-based courses, and look up grades and academic results.

Administrator / Academic Staff (Admin): The operational user group of the system.

- Role: Responsible for managing master data (adding new students, courses, faculties), configuring course registration periods, and inputting or adjusting grades.

External Systems:

Email System (SMTP Server):

- Purpose: The system utilizes this service to send automated notifications to users, including successful course registration confirmations, new grade alerts, and important academic reminders.
- Protocol: SMTP / API.

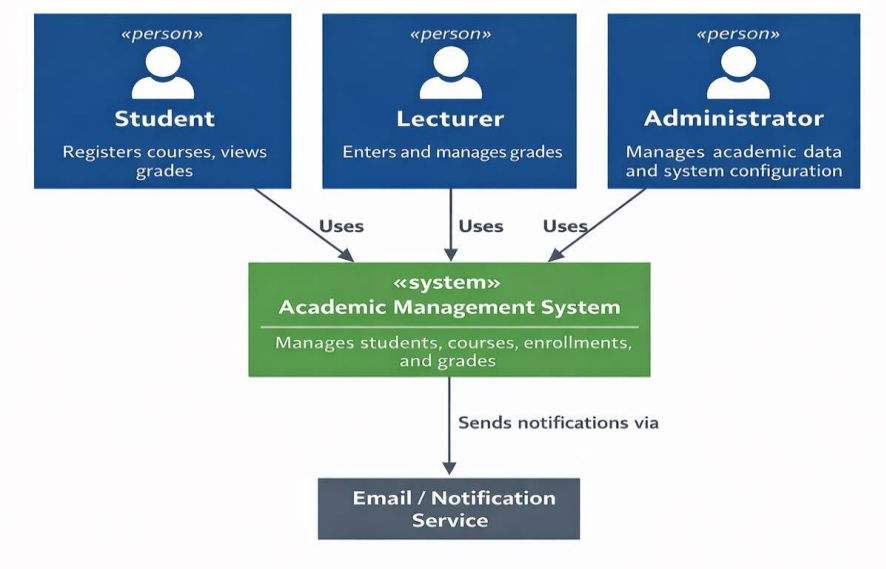


Figure C4 – System Context Diagram for Academic Management System

3.2 Architectural Pattern: Microservices Architecture

The Academic Management System is designed following the Microservices Architecture pattern. Instead of building a single monolithic application, the system is decomposed into a set of small, domain-oriented services, each responsible for a specific business capability such as authentication, student management, course management, enrollment processing, grading, and notification.

Each microservice:

- Runs as an independent Flask application on a dedicated port.
- Encapsulates its own business logic and exposes a well-defined RESTful API.
- Can be developed, tested, and deployed independently at the code level.

This architectural approach is well suited for the Academic Management domain due to the clear separation of responsibilities among functional areas (e.g., enrollment, grading, identity), which reduces coupling and improves long-term maintainability.

The key benefits of applying Microservices Architecture in this system include:

- Scalability (Design-Level): Services with high load potential, such as Enrollment Service or Grade Service, are designed to be independently scalable in future deployments when supported by containerization and orchestration technologies.

- Maintainability: Clear service boundaries allow teams to modify or extend one functional area without impacting others, simplifying maintenance and future enhancements.

- Fault Isolation (Partial): Failures in non-critical services (e.g., Email Service) do not block core academic functionalities.

However, shared infrastructure components such as the database and identity service remain critical dependencies.

In this system, supporting domain services such as Faculty Service, University Service, Major Service, and KKT Service are responsible for managing organizational structure and curriculum-related data, further demonstrating the domain-driven decomposition applied in the overall architecture.

3.6 Client-Side Architecture

The client side is implemented as a Single Page Application (SPA) using pure JavaScript (Vanilla JS). The frontend communicates directly with each microservice through REST APIs.

Main Responsibilities of the Frontend

- User authentication: managing login tokens and redirection.
- Dynamic UI rendering: Using DOM manipulation to switch views (Dashboard, Student List, Grade Input) without reloading the page.

- Sending HTTP requests to appropriate services:

+ Identity Service (Port 5004) for authentication.

+ Student Service (Port 5001) for profile management.

+ Course Service (Port 5002) and Enrollment Service (Port 5006) for registration logic.

+ Grade Service (Port 5003) for viewing and updating grades.

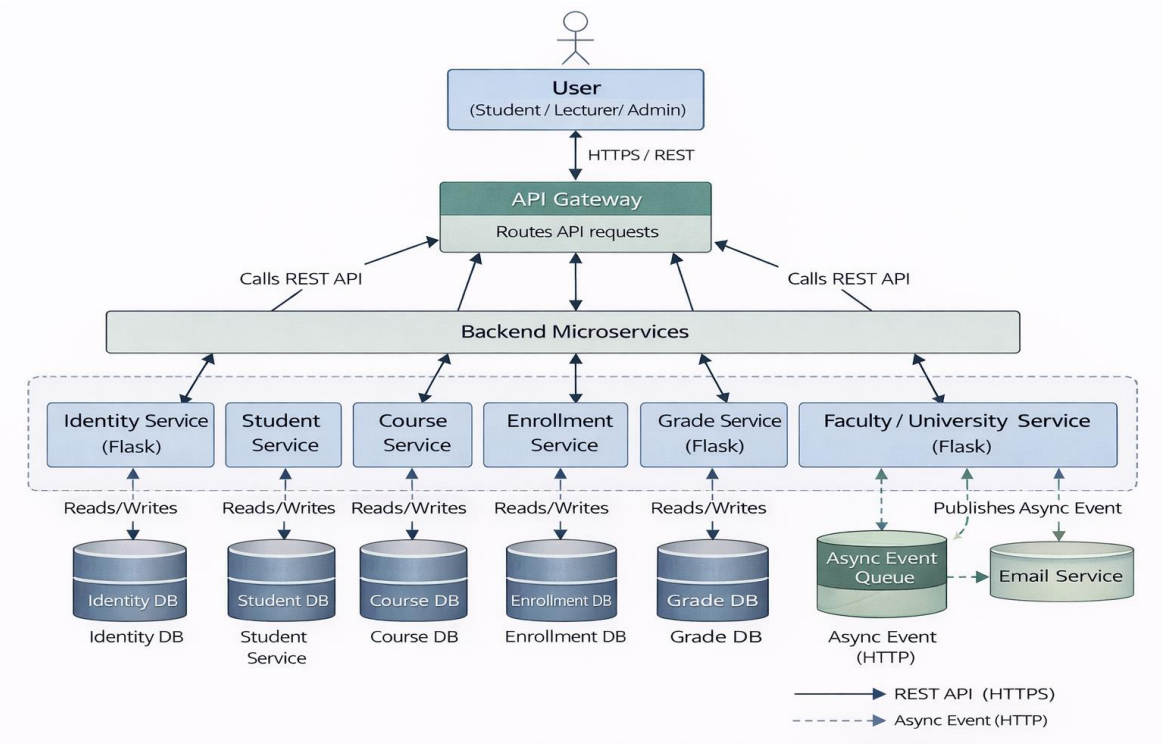
Communication Flow

- The user interacts with the HTML interface (e.g., clicks "Save Grade").

- JavaScript captures the event and sends an HTTP request (via fetch) to the corresponding microservice endpoint.

- The microservice processes the request and accesses the database.

- The response is returned in JSON format, and JavaScript updates the DOM to reflect changes immediately.



The figure shows the High-Level Communication Diagram of the system, highlighting interactions between the Frontend, backend microservices, databases, and the Email Service.

Summary: This architectural design combines Microservices Architecture with a lightweight JavaScript frontend and a centralized MySQL database. The design ensures modularity, scalability, and clear separation of concerns between presentation, business logic, and data layers.

4. Testing & Verification

4.1 Testing Strategy (Unit/Integration Test)

Due to the distributed nature of the Microservices architecture, the system adopts a multi-layer testing strategy to ensure that each service operates correctly on its own before being integrated into the overall system.

The testing strategy includes Unit Testing, Integration Testing, End-to-End Testing, and Functional Testing, covering both technical correctness and business requirements.

4.1.1 Unit Testing

Unit Testing focuses on verifying the smallest testable components within each Microservice, ensuring that internal logic and data processing are correct.

- ❖ Model Testing

This level tests the Entity classes defined in models.py (such as Student, Course, Grade).

- Verify that objects are correctly initialized with valid input data.
- Ensure entity fields are properly mapped to database attributes.
- Validate data transformation methods such as to_dict() to confirm JSON serialization works as expected.

- ❖ Repository Testing

This level tests the data access layer implemented in repository.py.

- Validate CRUD operations (INSERT, SELECT, UPDATE, DELETE) against the MySQL database.
- Ensure SQL queries are executed correctly via SQLAlchemy.
- Verify exception handling for database-related errors (e.g., connection failure, invalid queries).

4.1.2 Integration Testing

Integration Testing verifies the interaction between components within a service and between services through REST APIs and asynchronous communication.

- ❖ API Testing

API endpoints of each service are tested using Postman and cURL.

- HTTP methods tested: GET, POST, PUT, DELETE

- Objectives:
    - Verify correct HTTP status codes (200, 201, 400, 404, 500).
    - Validate the returned JSON structure and error messages.

Example:

Sending a POST /api/grades request with missing required fields should return:

+ HTTP 400 – Bad Request

+ Error message indicating invalid input data.

- ❖ Inter-service Communication Testing

This testing focuses on asynchronous communication between services.

- Scenario: Grade Service → Email Service
- When a lecturer finishes entering and locking grades, the Grade Service publishes an event to the Event Bus.
- The Email Service subscribes to this event and processes the email notification.

Verification method:

Check console logs of both services to confirm event publishing and consumption

5. Conclusion & Reflection

5.1 Lessons Learned

Through the implementation of the Academic Management System based on Microservices Architecture, the project team has gained valuable practical experience that goes beyond theoretical knowledge covered in class.

Firstly, the project helped the team clearly understand the differences between Monolithic and Microservices architectures. While Microservices offer significant advantages in scalability, independent deployment, and fault isolation, they also introduce complexity in service coordination, data consistency, and inter-service communication. Designing clear service boundaries (Identity, Student, Course, Grade, Enrollment, etc.) proved to be a critical architectural decision that directly impacted system maintainability and extensibility.

Secondly, the team gained hands-on experience with distributed system challenges, particularly in ensuring data consistency and fault tolerance. Scenarios such as course registration under high concurrency required careful handling to avoid issues like over-enrollment. This reinforced the importance of architectural patterns such as Database-per-Service, asynchronous processing, and circuit breaker mechanisms to prevent cascading failures when a service becomes unavailable.

Thirdly, the project highlighted the importance of non-functional requirements (NFRs) in architectural design. Performance, security, and reliability were not treated as afterthoughts but were explicitly translated into concrete design decisions, such as JWT-based authentication, role-based access control (RBAC), background email processing, and response-time constraints. This approach helped the team appreciate how quality attributes drive architecture selection and implementation strategies.

Finally, teamwork and collaboration were also key lessons learned. Dividing responsibilities among team members while maintaining consistent API standards and coding conventions required effective communication and documentation. The experience emphasized the value of clear interface contracts, proper logging, and structured testing to support teamwork in real-world software development.

5.2 Future Improvements

Although the system has achieved its primary objectives, several areas can be improved and extended in

future iterations.

Advanced Event-Driven Implementation: While RabbitMQ is currently implemented for Grade events, future iterations will expand its usage to all services (e.g., Enrollment, Identity) to fully replace direct HTTP calls for inter-service communication.

In terms of scalability and deployment, containerization using Docker and orchestration with Kubernetes would allow services to scale independently based on workload, especially during peak course registration periods. Additionally, implementing centralized monitoring and logging (e.g., Prometheus, Grafana, ELK stack) would enhance system observability and simplify maintenance in production environments.

Regarding security, further enhancements could include token refresh mechanisms, rate limiting at the API Gateway, and more advanced auditing features to track sensitive operations such as grade unlocking or Administrator data changes.

From a functional standpoint, the system can be extended to support additional features such as online payment integration for tuition fees, advanced academic analytics dashboards, and integration with existing university systems (e.g., Learning Management Systems – LMS).

In conclusion, this project serves as a solid foundation and a practical case study for applying Microservices Architecture to a real-world academic management problem. The lessons learned and proposed improvements provide a clear roadmap for future development and deeper exploration of modern distributed software architectures.