PHENIKAA UNIVERSITY

PHENIKAA SCHOOL OF COMPUTING

COURSE: SOFTWARE ARCHITECTURE

TOPIC:

DEVLOPMENT OF TRAINING PROGRAM MANAGEMENT SOFTWARE

REPORT: LAB 7 – EVENT-DRIVEN ARCHITECTURE (EDA) & INTEGRATION

Code Course : CSE703110-1-1-25

Class : N01

Instructor : THS. Vũ Quang Dũng

Group : 12

Members : 1. Đồng Đại Đạt : KTPM.EL2 – 23010877

2. Nguyễn Cao Hải : KTPM.EL2 – 23013124

3. Ngô Nhật Quang : KTPM.EL1 – 23010134

4. Nguyễn Nam Khánh : KTPM.EL2 – 23010771

Hanoi, November 2025

CONTENTS

1. Abstract

This laboratory explores the application of Event-Driven Architecture (EDA) to enable asynchronous communication between services in the Training Program Management Software. Instead of relying on synchronous service-to-service interactions, the system adopts a message-based communication model using RabbitMQ as a message broker and Celery as the asynchronous task framework.

A representative scenario is implemented in which a student enrollment action triggers an event that is published to a message queue. A separate Notification Service consumes this event and processes it independently, such as sending a confirmation email. Through this design, the lab demonstrates loose coupling, improved fault tolerance, and enhanced scalability, which are key architectural benefits of EDA.

2. Objectives

The objectives of this lab are:

+ To understand the principles of Event-Driven Architecture.
+ To distinguish between synchronous and asynchronous service communication.
+ To identify the roles of Producer, Consumer, and Message Broker.
+ To integrate RabbitMQ as a message broker in a service-oriented system.
+ To analyze the non-functional impacts of asynchronous integration.

3. System Context

The Training Program Management Software manages academic processes such as student enrollment, course registration, grading, and notifications. In previous labs, services interacted synchronously through REST APIs behind an API Gateway. While this approach is suitable for request-response workflows, it introduces tight temporal coupling between services.

Certain operations, such as sending notifications after a successful enrollment, do not require immediate responses and should not block core business logic. To address this

limitation, Lab 7 introduces an asynchronous communication mechanism based on Event-Driven Architecture.

4. Event-Driven Architecture Pattern

Event-Driven Architecture is an architectural style in which system components communicate by producing and consuming events. An event represents a significant change in system state and is emitted by a producer without knowledge of which components will react to it.

In contrast to synchronous REST-based communication, EDA decouples services in both space and time. Producers do not need to know the location, availability, or implementation details of consumers. Instead, a message broker acts as an intermediary, storing events until they are processed by interested consumers.

In this lab, the EDA pattern is applied to decouple the enrollment process from the notification process. When a student successfully enrolls in a course, the enrollment module emits an event describing this action. The Notification Service subscribes to such events and handles them asynchronously. This design prevents notification-related failures or delays from affecting the core enrollment workflow.

5. Architecture Design with EDA

5.1 Overall Architecture

The system architecture consists of the following components:

+ Producer: Enrollment module within the Django monolith
+ Message Broker: RabbitMQ
+ Consumer: Notification Service
+ Async Framework: Celery

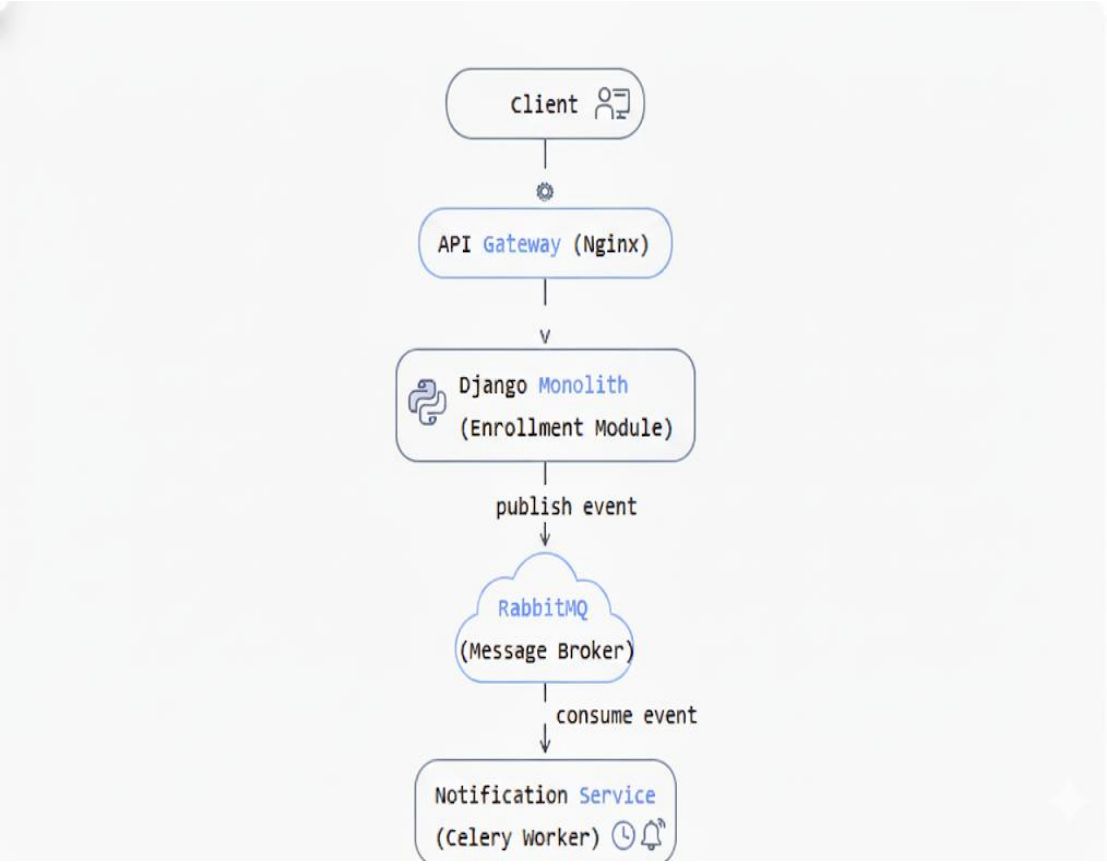

Figure 1. Event-Driven Architecture for asynchronous enrollment notification.

6. Event Flow Description

The event flow implemented in this lab is as follows:

1. A student submits a request to enroll in a course.

2. The monolithic backend validates the request and persists the enrollment data in the database

3. After a successful transaction, the system publishes an EnrollmentCreated event using Celery

4.Celery serializes the task and sends it to RabbitMQ

5.RabbitMQ stores the event in a dedicated queue

6.The Notification Service's Celery worker consumes the event

7.The Notification Service processes the event and sends a confirmation email (simulated)

8.If the Notification Service is unavailable, the event remains in the queue until the service resumes

7. Message Broker and Communication Model

RabbitMQ is used as the message broker to support reliable message delivery. The broker decouples producers and consumers by buffering events and ensuring that messages are not lost when consumers are temporarily unavailable.

Celery is integrated on both the producer and consumer sides to simplify interaction with RabbitMQ. Instead of manually managing AMQP connections, Celery abstracts message publishing and consumption through task definitions. This integration reduces boilerplate code and aligns well with the Python-based technology stack of the system

8. Non-Functional Requirement Analysis

8.1 Performance

By offloading notification processing to an asynchronous workflow, the enrollment operation completes faster from the client's perspective. The producer does not wait for the notification logic to execute, thereby reducing response time and improving user experience

8.2 Scalability

EDA enables independent scaling of producers and consumers. As enrollment volume increases, additional Notification Service workers can be deployed without modifying the enrollment logic. RabbitMQ efficiently distributes messages among available consumers, supporting horizontal scalability

8.3 Reliability and Fault Tolerance

The message broker acts as a buffer that isolates failures. If the Notification Service is temporarily unavailable, enrollment events are preserved in the queue and processed later. This prevents non-critical service failures from affecting core system functionality

8.4 Loose Coupling and Maintainability

The Enrollment module and Notification Service do not directly depend on each other. Changes to notification logic, delivery channels, or implementation details do not require modifications to the enrollment code. This loose coupling improves maintainability and supports future system evolution.

9. Testing and Validation

Testing is conducted by executing the enrollment workflow under different conditions:

+ Normal Operation: Both services running; enrollment triggers notification successfully..

+ Consumer Failure: Notification Service stopped; enrollment still succeeds and events accumulate in the queue.

+ Recovery Scenario: Notification Service restarted; queued events are processed correctly

10. Conclusion

Lab 7 successfully demonstrates the integration of asynchronous communication using Event-Driven Architecture within the Training Program Management Software. By introducing RabbitMQ and Celery, the system achieves loose coupling, improved fault tolerance, and enhanced scalability. This architectural approach prepares the system for future expansion and more complex asynchronous workflows.