YUMMY – ONLINE FOOD ORDERING AND DELIVERY SYSTEM

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

I. Summary

Lab 7 focuses on implementing the **Event-Driven Architecture (EDA)** pattern to enable asynchronous communication between microservices within the Yummy system. Instead of the Order Service waiting for the Notification Service to finish sending confirmation emails, we utilize **RabbitMQ** as a Message Broker to decouple these services. This approach significantly improves system performance, ensures non-blocking user experiences, and enhances reliability, as messages can be queued and processed even if the consumer service is temporarily unavailable.**Technology and tool installation**
The API Gateway for the Yummy system is built using the Node.js environment to handle concurrent requests efficiently.

II. Technology and Tool Installation

The implementation of this event-driven flow requires a message broker and specific libraries for the Node.js environment.

Tool	Purpose	Installation/Setup Guide
RabbitMQ	The Message Broker that stores and routes order events.	Running via Docker or CloudAMQP.
Node.js	Execution environment for the Producer and Consumer services.	Installed Node.js LTS version.
Dotenv	Manage environment variables (URLs, Ports) without hard-coding.	npm install dotenv

Table 7.1 List of supporting tools and libraries used in Lab 7.

III. Lab Specific Section: Event-Driven Implementation

1. Project Structure and Configuration

Two independent services were established to simulate the asynchronous workflow:

- Order Service (Event Producer): Located at src/backend/services/order-service/.
- Notification Service (Event Consumer): Located at src/backend/services/notification-service/.
- Shared Configuration: Both services connect to amqp://localhost using a dedicated queue named order_events.

2. The Order Service (Event Producer)

The Order Service is responsible for processing the checkout logic. Once an order is "placed," it does not call the notification logic directly. Instead, it creates an OrderPlaced event containing the orderId and customerEmail, then publishes this message to the RabbitMQ queue. This allows the Order Service to respond to the customer immediately.

3. Notification Service (Event Consumer)

The Notification Service constantly listens to the order_events queue. When a new message arrives:

1. It parses the order data.
2. It simulates the time-consuming process of sending an email or Firebase push notification (using a 1-second delay).
3. It acknowledges the message completion to RabbitMQ.

IV. Activity Practice: Testing Asynchronous Decoupling

Goal: Demonstrate that the Order Service is not blocked by the Notification Service's processing time. Step-by-Step Observation:

1. Start Consumer: The Notification Service terminal displays: [*] Waiting for OrderPlaced Events.
2. Run Producer: The Order Service terminal quickly prints: [x] Order Service: Sent order event for ID: 101. The process finishes instantly.
3. Process Verification: The Notification Service terminal receives the event and logs: [x] Notification Service: Received order 101. After a 1-second delay (simulating email delivery), it logs: [✓] Confirmation notification sent to: customer@example.com.

Conclusion of Test: Even if the Notification Service takes a long time to send emails or is temporarily shut down, the Order Service remains fully functional. The messages stay safely in the RabbitMQ queue until the consumer is ready to process them.

V. Modeling Artifact: Event-Driven Logic Flow

The system follows a decoupled execution flow:

1. Client: Places an order via the API.
2. Order Service: Saves order to DB and publishes OrderPlaced event to RabbitMQ.
3. RabbitMQ: Holds the event in the order_events queue.
4. Notification Service: Picks up the event from the queue and handles the external delivery logic asynchronously.

VI. Conclusion

The implementation of Event-Driven Architecture in Lab 7 successfully addresses the latency issues of the Yummy platform. By using RabbitMQ as a mediator, we achieved high scalability and fault tolerance. This ensures that critical business processes (like placing an order) are never delayed by secondary tasks (like sending notifications), providing a seamless experience for the end-user.