Lab 7: Event-Driven Architecture (EDA) & Integration

This lab introduces the **Event-Driven Architecture (EDA)** pattern, focusing on **asynchronous communication** between microservices. We will use a message broker to decouple the **Order Service** (Producer) from the **Notification Service** (Consumer), specifically for the task of sending a confirmation email after an order is placed.

Objectives

1. Understand the concepts of **Producers, Consumers, and Message Brokers**.
2. Install and set up a local message broker (using **RabbitMQ**).
3. Implement a simple **Order Service** that acts as the event **Producer**.
4. Implement a **Notification Service** that acts as the event **Consumer**.
5. Demonstrate the **decoupled nature** of the two services.

Technology & Tool Installation

We will use **Python** and the **Pika** library to interact with the **RabbitMQ** message broker.

Tool	**Purpose**	**Installation/Setup Guide**
RabbitMQ	The **Message Broker** that holds and routes events.	**Option 1 (Recommended):** Use Docker. Install **Docker Desktop**, then run: docker run -d --hostname rabbitmq-host --name rabbitmq -p 5672:5672 -p 15672:15672 rabbitmq:3-management
Python 3.x	Core programming language.	Ensure Python 3 is installed.
Pika	Python client library for connecting to RabbitMQ.	Run: pip install pika
Two Separate Terminals	To run the Producer (Order Service) and Consumer	N/A

Tool	Purpose	Installation/Setup Guide
	(Notification Service) concurrently.	

Activity Practice 1: Setup and Broker Connection

Goal: Ensure RabbitMQ is running and set up the basic connection configuration.

Step-by-Step Instructions

1. **Start RabbitMQ:** Run the Docker command above. Verify the broker is running by opening the management console in your browser: http://localhost:15672 (default guest/guest login).
2. **Create Project Structure:**

Bash

```
mkdir shopsphere_eda

cd shopsphere_eda

python -m venv venv

source venv/bin/activate

pip install pika

touch order_service_producer.py

touch notification_service_consumer.py
```

3. **Define Broker Parameters:** In both Python files, define the connection details.

Configuration (Add to both files):

Python

```
# Connection parameters for RabbitMQ

RABBITMQ_HOST = 'localhost'

QUEUE_NAME = 'order_events'
```

Activity Practice 2: Order Service (Event Producer)

Goal: Simulate the Order Service completing an order and publishing an OrderPlacedEvent to the message queue.

Step-by-Step Instructions & Coding Guide

1. **Implement Producer Logic:** The producer connects to RabbitMQ, ensures the queue exists, and publishes a JSON message (the event).

File: order_service_producer.py

Python

```
import pika

import json

import time


RABBITMQ_HOST = 'localhost'

QUEUE_NAME = 'order_events'


def publish_order_placed_event(order_data):

    """Connects to RabbitMQ and publishes the event."""

    try:

        # Connect to the broker

        connection =
pika.BlockingConnection(pika.ConnectionParameters(host=RABBITMQ_HOST))

        channel = connection.channel()


        # Declare the queue (idempotent - creates if it doesn't exist)

        channel.queue_declare(queue=QUEUE_NAME)


        # Convert Python dict to JSON string (the event payload)
```

```
        message = json.dumps(order_data)

        # Publish the message to the queue
        channel.basic_publish(
            exchange='',        # Default exchange (sends to the queue name)
            routing_key=QUEUE_NAME,
            body=message
        )
        print(f" [x] Order Service published event for Order ID: {order_data['order_id']}")

        connection.close()
    except pika.exceptions.AMQPConnectionError as e:
        print(f" [!] Connection Error: Could not connect to RabbitMQ. Is it running? {e}")

if __name__ == '__main__':
    print("Order Service is starting...")
    # Simulate placing 3 orders every few seconds
    for i in range(1, 4):
        order_info = {
            "order_id": f"ORD-{i:03}",
            "customer_email": f"user{i}@example.com",
            "timestamp": time.time()
        }
        publish_order_placed_event(order_info)
        time.sleep(2)
```

Activity Practice 3: Notification Service (Event Consumer)

Goal: Implement the Notification Service. It should listen continuously for new events from the queue and process them (simulating sending an email).

Step-by-Step Instructions & Coding Guide

1. **Implement Consumer Logic:** The consumer connects, registers a callback function, and enters a continuous listening loop.

File: notification_service_consumer.py

Python

```
import pika
import json
import time

RABBITMQ_HOST = 'localhost'
QUEUE_NAME = 'order_events'

def callback(ch, method, properties, body):
    """This function is called when a message is received."""
    try:
        order_data = json.loads(body)
        order_id = order_data.get('order_id')
        email = order_data.get('customer_email')

        print(" [x] Notification Service received an event.")

        # Simulate heavy email sending logic (Processing time)
        time.sleep(1)
```

```
        # Core Business Logic of the Consumer
        print(f" [✓] SENT CONFIRMATION: Order {order_id} confirmed for email: {email}")

        # Acknowledge the message (tells RabbitMQ the message was processed successfully)
        ch.basic_ack(delivery_tag=method.delivery_tag)

    except json.JSONDecodeError:
        print(f" [!] Error decoding JSON: {body}")
        ch.basic_reject(delivery_tag=method.delivery_tag, requeue=False) # Reject bad message

if __name__ == '__main__':
    try:
        print("Notification Service is connecting...")
        connection =
pika.BlockingConnection(pika.ConnectionParameters(host=RABBITMQ_HOST))
        channel = connection.channel()

        # Ensure the queue exists
        channel.queue_declare(queue=QUEUE_NAME)

        # Pre-fetch limit (optional but good practice: prevents overwhelming the consumer)
        channel.basic_qos(prefetch_count=1)

        # Start consuming messages, passing the callback function
        channel.basic_consume(queue=QUEUE_NAME, on_message_callback=callback)
```

```
    print(' [*] Waiting for OrderPlacedEvents. To exit press CTRL+C')
    channel.start_consuming()

except pika.exceptions.AMQPConnectionError as e:
    print(f" [!] Connection Error: Could not connect to RabbitMQ. Is it running? {e}")
except KeyboardInterrupt:
    print('Consumer shutting down.')
```

Activity Practice 4: Testing Asynchronous Decoupling

Goal: Observe the decoupled execution of the two services.

Step-by-Step Instructions

1. **Start the Consumer (Notification Service):**
    - **Action:** Open the **first terminal** in the shopsphere_eda directory.
    - **Command:** python notification_service_consumer.py
    - **Observation:** The consumer should start and print: [*] Waiting for OrderPlacedEvents. To exit press CTRL+C.
2. **Run the Producer (Order Service):**
    - **Action:** Open the **second terminal** in the shopsphere_eda directory.
    - **Command:** python order_service_producer.py
    - **Observation (Producer Terminal):** It should quickly print the three "Order Service published event..." lines (since publishing is very fast).
    - **Observation (Consumer Terminal):** The consumer will receive and process the events one by one, with a 1-second delay between each, demonstrating that the time taken to send the email (1 second) **did not block** the Order Service from completing the order and publishing the events.

Conclusion: This demonstrates that the Order Service is decoupled from the Notification Service. If the Notification Service were down, the Order Service would still complete the

transaction and the messages would simply wait in the RabbitMQ queue until the consumer restarts, ensuring **reliability and non-blocking operation**.