Slide Deck: Lecture 7 - Event-Driven Architecture (EDA) & Decoupling

1. Title Slide: Asynchronous Communication

- **Title:** Software Architecture: Lecture 7
- **Subtitle:** Event-Driven Architecture (EDA) & Decoupling
- **Focus:** Implementing **asynchronous communication** to achieve the **Fault Isolation ASR** for ShopSphere.

---

2. Recap: The Coupling Problem

- **Synchronous Flaw:** In Lectures 4-6, if the Order Service called the Notification Service directly (HTTP/REST), the Order transaction would **block** until the email was sent.
- **ASR 2 (Fault Isolation):** We mandated that failure in the Notification System must **not prevent** core order placement.
- **Solution:** We must use an **Event-Driven Architecture (EDA)** via a Message Broker.

---

3. Knowledge Base: Event-Driven Architecture (EDA)

- **Definition:** An architectural pattern based on the production, detection, consumption of, and reaction to **events**.
- **Core Components:**
    1. **Event (Message):** A record of something that happened (e.g., OrderPlaced).
    2. **Producer (Publisher):** The service that creates and sends the event (e.g., Order Service).
    3. **Consumer (Subscriber):** The service that reacts to the event (e.g., Notification Service).
    4. **Message Broker (Queue):** The intermediary that reliably buffers and routes the events.

---

4. Knowledge Base: The Message Broker (RabbitMQ)

- **Role:** The durable intermediary.

- **Function:** It holds messages until a Consumer is ready to process them.
- **Key Benefit: Temporal Decoupling**. The Producer doesn't need to know if the Consumer is online *now*. If the Consumer is down, the message waits safely in the queue.

---

5. Proof of Concept: Asynchronous Flow

- **Scenario:** Customer places an order.
- **The Flow:**
    1. **Order Service (Producer)** completes DB transaction.
    2. Producer sends OrderPlacedEvent to **Broker**.
    3. Producer returns **HTTP 200 OK** to the client (Order is done!).
    4. **Broker** holds the message.
    5. Later, **Notification Service (Consumer)** picks up the message and sends the email.
- **Result:** The customer receives a confirmation instantly, even if the email system takes 5 seconds to process.

---

6. Activity 1: Tool Installation and Setup

- **Goal:** Set up the Message Broker locally using Docker (Recommended).
- **Installation Steps (Broker):**
    1. Ensure **Docker Desktop** is installed and running.
    2. Run the RabbitMQ image: docker run -d --hostname rabbitmq-host --name rabbitmq -p 5672:5672 rabbitmq:3-management
- **Installation Steps (Code):**
    1. mkdir shopsphere_eda
    2. cd shopsphere_eda
    3. pip install pika (Python client for RabbitMQ).

---

7. Activity 2: Designing the Event

- **Goal:** Define the minimal data required for the event payload.
- **Event Name:** OrderPlacedEvent
- **Payload (JSON format):**

JSON

```
{
  "order_id": "ORD-123",
  "customer_email": "user@example.com",
  "timestamp": 1700000000
}
```

---

8. Knowledge Base: Producer Logic (Pika)

- **Key Pika Steps:**
    1. Establish a **Connection** to the broker.
    2. Create a **Channel** (the virtual circuit).
    3. Declare the **Queue** (ensures it exists).
    4. Use channel.basic_publish to send the message.

---

9. Code Step 1: Order Service (Producer) Setup

- **File: order_service_producer.py**

Python

```
import pika, json, time
RABBITMQ_HOST = 'localhost'
QUEUE_NAME = 'order_events'


def publish_event(order_data):
```

```
    try:
        connection =
pika.BlockingConnection(pika.ConnectionParameters(host=RABBITMQ_HOST))
        channel = connection.channel()
        channel.queue_declare(queue=QUEUE_NAME) # 1. Declare Queue


        message = json.dumps(order_data)


        # 2. Publish the Message
        channel.basic_publish(
            exchange='',
            routing_key=QUEUE_NAME,
            body=message
        )
        print(f" [x] Order Service published event for {order_data['order_id']}")
        connection.close()
    except pika.exceptions.AMQPConnectionError:
        print(" [!] Connection Error: RabbitMQ not running.")
```

---

10. Activity 3: Implementing the Producer Test

- **Goal:** Simulate the Order Service completing three orders rapidly.
- **File: order_service_producer.py (Main Block)**

Python

```
if __name__ == '__main__':
    print("Order Service is starting...")
    for i in range(1, 4):
```

```
    order_info = {
        "order_id": f"ORD-{i:03}",
        "customer_email": f"user{i}@example.com"
    }
    publish_event(order_info)
    time.sleep(0.1) # Simulate quick processing time
```

---

11. Knowledge Base: Consumer Logic (Pika)

- **Key Pika Steps:**
    1. Establish Connection and Channel.
    2. Define a **Callback Function** that processes the received message.
    3. Use channel.basic_consume to register the callback.
    4. Use channel.start_consuming to enter the infinite listening loop.
    5. Crucial: Send an **Acknowledgment (basic_ack)** after successful processing.

---

12. Code Step 2: Notification Service (Consumer) Callback

- **File: notification_service_consumer.py**

Python

```
def callback(ch, method, properties, body):
    order_data = json.loads(body)
    order_id = order_data.get('order_id')
    email = order_data.get('customer_email')

    print(f" [x] Received event for Order ID: {order_id}")

    # Simulate high latency task (Sending Email)
```

```
    time.sleep(3)

    # Core Business Logic
    print(f" [✓] SENT CONFIRMATION: Order {order_id} to {email}")

    # Acknowledge the message (tells the broker the message is safe to delete)
    ch.basic_ack(delivery_tag=method.delivery_tag)
```

---

13. Code Step 3: Notification Service (Consumer) Listener

- **File: notification_service_consumer.py (Main Block)**

```
Python
if __name__ == '__main__':
    try:
        connection = pika.BlockingConnection(pika.ConnectionParameters(host=RABBITMQ_HOST))
        channel = connection.channel()
        channel.queue_declare(queue=QUEUE_NAME)

        # Register the callback function
        channel.basic_consume(queue=QUEUE_NAME, on_message_callback=callback, auto_ack=False)

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

    except pika.exceptions.AMQPConnectionError:
```

```
    print(" [!] Connection Error: RabbitMQ not running.")
except KeyboardInterrupt:
    print('Consumer shutting down.')
```

---

14. Activity 4: Proof of Concept - Decoupling Test

- **Goal:** Run both services concurrently to observe temporal decoupling.
- **Action Steps:**
    1. **Terminal 1:** Run Consumer: python notification_service_consumer.py (Must run first).
    2. **Terminal 2:** Run Producer: python order_service_producer.py
    3. Wait and observe output in both terminals.

---

15. Verification: Analyzing Terminal Output

- **Producer Terminal Output (Order Service):** The three "published event" messages appear almost **instantaneously** (0.1s delay between each).
- **Consumer Terminal Output (Notification Service):** The "SENT CONFIRMATION" messages appear with a **3-second delay** between each, processing the queue sequentially.
- **Conclusion:** The Order Service was **not blocked** by the 3-second email latency, successfully meeting the **Fault Isolation ASR (ASR 2)**.

---

16. Architectural Significance: Fault Tolerance

- **What if the Consumer was NOT running?** The Producer would still publish events, and they would pile up safely in the RabbitMQ queue.
- **Recovery:** When the Notification Service restarts (e.g., after maintenance or a crash), it immediately begins processing the backlog of messages from the queue.
- **Result:** Zero data loss, maximum resilience for the core business transaction (placing the order).

---

17. EDA Trade-Offs

- **Pro (Resilience):** Guaranteed delivery, temporal decoupling, excellent fault tolerance, and clear boundary between services.
- **Con (Complexity):** Introduces a new, critical dependency (the Broker). Requires dealing with message guarantees (at-least-once, exactly-once) and potential **Message Idempotency** (handling the same message twice without error).
- **Trade-off:** We traded simple, direct calls for the complexity of a **distributed transaction** to gain crucial fault isolation.

---

18. Integrating EDA with Synchronous Calls

- **The Mix:** In a true Microservices architecture, you need both:
    - **Synchronous (REST):** For immediate, transactional responses (e.g., Cart $\rightarrow$ Product Service for price).
    - **Asynchronous (EDA):** For notifications, logging, long-running processes, and background tasks (e.g., Order $\rightarrow$ Notification).

---

19. Summary of Deliverables for Submission

- **Document 1:** Code snippet of the **Producer's basic_publish** command.
- **Document 2:** Code snippet of the **Consumer's callback function** showing the time.sleep(3) (simulated latency) and the ch.basic_ack call.
- **Document 3:** Terminal output from both the Producer and Consumer running simultaneously, clearly showing the **instantaneous Producer output** followed by the **delayed Consumer processing**.

---

20. Q&A and Next Steps

- **Questions?** (Review the difference between temporal and structural coupling).
- **Pre-work:** Research **UML Deployment Diagrams** and the **ATAM** (Architecture Trade-off Analysis Method).

- **Next Lecture: Lecture 8: Deployment View & Quality Attribute Analysis (ATAM)**. We finalize documentation and evaluate the entire architecture.