Spring Transactional Kafka Demo

This project demonstrates a robust, transactional "Staging -> Public" queue pattern using Spring Kafka.

Project Premise: Why Staging?

You might wonder: *"Why not publish directly to the Public topic?"*

The Problem: We CANNOT control the configuration of downstream consumers on the Public topic.

- Many legacy or external consumers use the default isolation level: `read_uncommitted`.
- If we publish a transaction that later fails (rolls back), these consumers would see "Dirty Read" data (aborted messages).

The Solution: The "Sanitizer Pattern" *AKA: Transactional Forwarding / Anti-Corruption Layer*

> **Disclaimer**: *"Sanitizer Pattern"* is a descriptive term coined for this project to intuitively describe the function of this service. In professional contexts, this is known as an **Anti-Corruption Layer** implementing **Transactional Consume-Transform-Produce**.

1. **Staging Queue (Dirty Water)**: Internal topic. Contains mixed data: some committed, some aborted/failed.

2. **Sanitizer Service (The Filter)**: Our internal consumer. Uses `isolation-level: read_committed` to filter out the "dirty" aborted messages.
3. **Public Queue (Clean Water)**: External topic. Guaranteed to contain *only* valid, committed data.

Result: Downstream consumers on the Public Queue are protected from rollback noise and partial transactions, regardless of their own configuration.

Key Concepts

1. Scaling & Load Balancing

The application is designed to be horizontally scalable. You can run multiple instances (pods) of this application, and Kafka will automatically distribute the work.

How it works:

- **Topics**: `stagingA` and `stagingB` are configured with **3 partitions** each.
- **Consumer Group**: All application instances join the same Consumer Group: `staging-to-public-group`.
- **Distribution**: Kafka ensures that each partition is assigned to **only one** instance at a time.

Example with 2 Pods: If you scale to 2 replicas:

- **Pod 1** processes: `stagingA-0`, `stagingA-1`, `stagingB-0`
- **Pod 2** processes: `stagingA-2`, `stagingB-1`, `stagingB-2`

Result: Throughput doubles, but **no duplicate messages**.

2. Consumer Offsets & Restart Behavior

You might wonder: *"If a new pod starts, will it re-read old messages?"*

Answer: No.

Kafka tracks the **Consumer Offset** (the index of the last read message) for each Consumer Group.

- **Scenario**: Pod 1 crashes after processing message #100.
- **Recovery**: Pod 2 starts up and joins `staging-to-public-group`.
- **Action**: Pod 2 asks Kafka, "Where did we leave off?". Kafka replies, "Offset 100".
- **Result**: Pod 2 starts reading from message **#101**.

Note: `auto-offset-reset: earliest` only applies if the Consumer Group is brand new and has NO history. Once a group exists, its history is the source of truth.

3. Why Partitions?

You might ask: *"Why split a topic into 3 partitions? Why not just 1?"*

Answer: Speed (Parallelism).

Partitions are the **Unit of Parallelism** in Kafka.

- **Rule**: A single partition can only be consumed by **ONE** consumer instance at a time (to preserve order).
- **Consequence**: If you have 1 partition and 10 app instances (Pods), **9 Pods will sit idle**.
- **Solution**: With 3 partitions, you can have up to 3 active Pods working simultaneously.

Summary: More Partitions = Higher Max Concurrency.

4. FAQ: What if I have 1 Partition and 2 Pods?

It will **work**, but it won't **scale**.

- **Pod 1**: Will process ALL messages (Active).
- **Pod 2**: Will do NOTHING (Idle/Standby).

Kafka will not split the work because splitting a single partition would break the "Order Guarantee". To use both Pods, you must have at least 2 Partitions.

Nuance for IO-Bound Workloads: If your processing is very cheap (just forwarding), 1 Pod might easily handle the load. However, having 3 partitions allows all 3 pods to be **Active (Warm)**.

- **1 Partition**: Pod 2 is "Cold" (Idle). If Pod 1 dies, there is a rebalance delay before Pod 2 takes over.
- **3 Partitions**: All pods are processing. If Pod 1 dies, the others just take a bit more load immediately.

5. FAQ: Can I mix Object Types in one Topic?

Best Practice: NO. (1 Topic = 1 Payload Type)

Technically, Spring Kafka *can* support mixed types using Headers (`__TypeId__`), but it is fragile.

- **Problem**: If you push an `Orange` object to a topic where consumers expect `Apple`, the deserializer will crash or produce garbage data.
- **Recommendation**: Create separate topics for separate data types (e.g., `staging-users` vs `staging-orders`).

Configuration Explained (`application.yml`)

Here is a line-by-line explanation of the critical Kafka settings:

```
spring:
  kafka:
    bootstrap-servers: kafka:9092  # Address of the Kafka broker

    producer:
      # Serializers convert Java Objects to bytes for the network
      key-serializer: org.apache.kafka.common.serialization.StringSerializer
      value-serializer: org.springframework.kafka.support.serializer.JsonSerializer

      # [CRITICAL] Enables Transactions on the Producer
      # This prefix tells Spring to manage a pool of transactional IDs (e.g., tx-0,
tx-1)
      transaction-id-prefix: tx-

      # Durability: Producer waits for acknowledgement from ALL in-sync replicas
      # This ensures data is definitely saved to disk before continuing
      acks: all

      properties:
        # Ensures exactly-once delivery within a producer session (prevents dupes
```

```
if network retries)
        enable.idempotence: true

    consumer:
      # Deserializers convert bytes back to Java Objects
      key-deserializer: org.apache.kafka.common.serialization.StringDeserializer
      value-deserializer:
org.springframework.kafka.support.serializer.JsonDeserializer

      # Start reading from the BEGINNING if we have no stored offset
      auto-offset-reset: earliest

      # [CRITICAL] Transactional Isolation
      # "read_committed" = Hide aborted transactions (messages from failed flows)
      # "read_uncommitted" (default) = Show everything, including rolled-back dirty
data
      isolation-level: read_committed

      # We disable auto-commit because the Transaction Manager handles offsets
automatically
      # The offset is committed only when the transaction commits
      enable-auto-commit: false

      properties:
        # Security: Allow deserialization of classes in this package
        spring.json.trusted.packages: com.example.kafka.service
```