Spring Kafka Transactions: Part 3 - Outbox Processor, Best Practices & Production Guide

Final part of the comprehensive guide covering outbox event processing, comparisons, best practices, CLI operations, and version highlights.

Outbox Event Processor Implementation

Simple Explanation: The Outbox Event Processor is a separate component that monitors the outbox table for new events and publishes them to Kafka. It ensures reliable, at-least-once delivery of events while maintaining the transactional consistency provided by the outbox pattern.

Advanced Outbox Event Processor

```
/**
 * Comprehensive outbox event processor with retry and monitoring
 */
@Component
@lombok.extern.slf4j.Slf4j
public class OutboxEventProcessor {

    @Autowired
    private OutboxEventRepository outboxEventRepository;

    @Autowired
    private KafkaTemplate<String, Object> kafkaTemplate;

    @Autowired
    private ObjectMapper objectMapper;

    @Autowired
    private MeterRegistry meterRegistry;

    private final int maxRetryAttempts = 5;
    private final Duration processingInterval = Duration.ofSeconds(10);
    private final int batchSize = 100;

    /**
     * Scheduled outbox event processing with batch optimization
     */
    @Scheduled(fixedDelayString = "#
{@outboxProcessorConfig.getProcessingIntervalMs()}")
    @Transactional
    public void processOutboxEvents() {

        log.debug("Starting outbox event processing batch");

        Timer.Sample sample = Timer.start(meterRegistry);
```

```
        try {
            // Fetch pending events in batches
            Pageable pageable = PageRequest.of(0, batchSize);
            List<OutboxEvent> pendingEvents = outboxEventRepository

.findByStatusOrderByCreatedAt(OutboxEvent.OutboxEventStatus.PENDING, pageable);

            if (pendingEvents.isEmpty()) {
                log.trace("No pending outbox events to process");
                return;
            }

            log.info("Processing outbox events batch: size={}",
pendingEvents.size());

            int successCount = 0;
            int failureCount = 0;

            // Process each event
            for (OutboxEvent event : pendingEvents) {
                try {
                    boolean processed = processOutboxEvent(event);
                    if (processed) {
                        successCount++;
                    } else {
                        failureCount++;
                    }
                } catch (Exception e) {
                    log.error("Error processing outbox event: eventId={}",
event.getId(), e);
                    failureCount++;
                }
            }

            // Update metrics

meterRegistry.counter("outbox.events.processed.success").increment(successCount);

meterRegistry.counter("outbox.events.processed.failure").increment(failureCount);
            meterRegistry.gauge("outbox.events.pending.count",

outboxEventRepository.countByStatus(OutboxEvent.OutboxEventStatus.PENDING));

            log.info("Outbox event processing completed: success={}, failures={}",
successCount, failureCount);

        } catch (Exception e) {
            log.error("Error in outbox event processing batch", e);
            meterRegistry.counter("outbox.events.batch.error").increment();

        } finally {
            sample.stop(Timer.builder("outbox.events.processing.duration")
                .register(meterRegistry));
```

```
        }
    }

    /**
     * Process individual outbox event with retry logic
     */
    @Transactional
    public boolean processOutboxEvent(OutboxEvent event) {

        log.debug("Processing outbox event: eventId={}, eventType={}",
event.getId(), event.getEventType());

        try {
            // Mark as processing
            event.setStatus(OutboxEvent.OutboxEventStatus.PROCESSING);
            event.setProcessingAttempts(event.getProcessingAttempts() + 1);
            outboxEventRepository.save(event);

            // Deserialize event data
            Object eventData = deserializeEventData(event.getEventData(),
event.getEventType());

            // Create Kafka producer record
            ProducerRecord<String, Object> producerRecord = new ProducerRecord<>(
                event.getKafkaTopic(),
                event.getKafkaKey(),
                eventData
            );

            // Add headers for traceability
            addEventHeaders(producerRecord, event);

            // Send to Kafka
            ListenableFuture<SendResult<String, Object>> future =
kafkaTemplate.send(producerRecord);

            // Wait for result with timeout
            SendResult<String, Object> result = future.get(30, TimeUnit.SECONDS);

            // Mark as processed
            event.setStatus(OutboxEvent.OutboxEventStatus.PROCESSED);
            event.setProcessedAt(Instant.now());
            event.setLastError(null);
            outboxEventRepository.save(event);

            log.info("Outbox event published successfully: eventId={}, topic={},
partition={}, offset={}",
                event.getId(), event.getKafkaTopic(),
                result.getRecordMetadata().partition(),
                result.getRecordMetadata().offset());

            return true;

        } catch (Exception e) {
```

```
            log.error("Failed to process outbox event: eventId={}", event.getId(),
e);

            // Handle failure
            return handleEventProcessingFailure(event, e);
        }
    }

    /**
     * Handle event processing failures with retry logic
     */
    private boolean handleEventProcessingFailure(OutboxEvent event, Exception
exception) {

        String errorMessage = exception.getMessage();
        if (errorMessage != null && errorMessage.length() > 500) {
            errorMessage = errorMessage.substring(0, 500);
        }

        event.setLastError(errorMessage);

        if (event.getProcessingAttempts() >= maxRetryAttempts) {
            // Max retries exceeded - mark as failed
            event.setStatus(OutboxEvent.OutboxEventStatus.FAILED);
            outboxEventRepository.save(event);

            log.error("Outbox event failed after {} attempts: eventId={}, error=
{}",
                maxRetryAttempts, event.getId(), errorMessage);

            // Send to dead letter queue or alert system
            handleFailedEvent(event, exception);

            return false;

        } else {
            // Reset to pending for retry
            event.setStatus(OutboxEvent.OutboxEventStatus.PENDING);
            outboxEventRepository.save(event);

            log.warn("Outbox event will be retried: eventId={}, attempt={}, error=
{}",
                event.getId(), event.getProcessingAttempts(), errorMessage);

            return false;
        }
    }

    /**
     * Retry failed events that haven't exceeded max attempts
     */
    @Scheduled(fixedDelayString = "#
{@outboxProcessorConfig.getRetryIntervalMs()}")
    @Transactional
```

```
    public void retryFailedEvents() {

        log.debug("Starting failed event retry processing");

        try {
            Pageable pageable = PageRequest.of(0, batchSize / 2); // Smaller batch
for retries
            List<OutboxEvent> failedEvents = outboxEventRepository
                .findFailedEventsForRetry(maxRetryAttempts, pageable);

            if (failedEvents.isEmpty()) {
                log.trace("No failed events to retry");
                return;
            }

            log.info("Retrying failed outbox events: count={}",
failedEvents.size());

            for (OutboxEvent event : failedEvents) {
                try {
                    // Add exponential backoff delay
                    long delayMs =
calculateRetryDelay(event.getProcessingAttempts());
                    if
(event.getCreatedAt().plus(Duration.ofMillis(delayMs)).isAfter(Instant.now())) {
                        continue; // Too early to retry
                    }

                    processOutboxEvent(event);

                } catch (Exception e) {
                    log.error("Error retrying failed event: eventId={}",
event.getId(), e);
                }
            }

        } catch (Exception e) {
            log.error("Error in failed event retry processing", e);
        }
    }

    /**
     * Cleanup processed events to prevent table growth
     */
    @Scheduled(cron = "#{@outboxProcessorConfig.getCleanupCron()}")
    @Transactional
    public void cleanupProcessedEvents() {

        log.info("Starting cleanup of processed outbox events");

        try {
            // Delete events processed more than configured retention period ago
            Instant cutoffTime = Instant.now().minus(Duration.ofDays(7)); // 7
days retention
```

```
            int deletedCount =
outboxEventRepository.deleteProcessedEventsOlderThan(cutoffTime);


meterRegistry.counter("outbox.events.cleanup.deleted").increment(deletedCount);

            log.info("Cleanup completed: deleted {} processed events older than
{}",
                deletedCount, cutoffTime);

        } catch (Exception e) {
            log.error("Error during outbox event cleanup", e);
            meterRegistry.counter("outbox.events.cleanup.error").increment();
        }
    }

    /**
     * Monitor outbox table health and metrics
     */
    @Scheduled(fixedDelayString = "#
{@outboxProcessorConfig.getHealthCheckIntervalMs()}")
    public void monitorOutboxHealth() {

        try {
            // Count events by status
            long pendingCount =
outboxEventRepository.countByStatus(OutboxEvent.OutboxEventStatus.PENDING);
            long processingCount =
outboxEventRepository.countByStatus(OutboxEvent.OutboxEventStatus.PROCESSING);
            long failedCount =
outboxEventRepository.countByStatus(OutboxEvent.OutboxEventStatus.FAILED);
            long processedCount =
outboxEventRepository.countByStatus(OutboxEvent.OutboxEventStatus.PROCESSED);

            // Update metrics
            meterRegistry.gauge("outbox.events.pending.count", pendingCount);
            meterRegistry.gauge("outbox.events.processing.count",
processingCount);
            meterRegistry.gauge("outbox.events.failed.count", failedCount);
            meterRegistry.gauge("outbox.events.processed.count", processedCount);

            // Check for concerning conditions
            if (pendingCount > 1000) {
                log.warn("High number of pending outbox events: {}",
pendingCount);
            }

            if (failedCount > 100) {
                log.warn("High number of failed outbox events: {}", failedCount);
            }

            if (processingCount > 50) {
                log.warn("High number of events stuck in processing: {}",
```

```
processingCount);
            }

            log.debug("Outbox health check: pending={}, processing={}, failed={},
processed={}",
                pendingCount, processingCount, failedCount, processedCount);

        } catch (Exception e) {
            log.error("Error during outbox health monitoring", e);
        }
    }

    // Helper methods
    private Object deserializeEventData(String eventData, String eventType) throws
Exception {

        switch (eventType) {
            case "OrderCreated":
                return objectMapper.readValue(eventData, OrderCreatedEvent.class);
            case "OrderStatusChanged":
                return objectMapper.readValue(eventData,
OrderStatusChangedEvent.class);
            case "PaymentConfirmed":
                return objectMapper.readValue(eventData,
PaymentConfirmedEvent.class);
            case "ShippingRequested":
                return objectMapper.readValue(eventData,
ShippingRequestedEvent.class);
            case "RefundRequested":
                return objectMapper.readValue(eventData,
RefundRequestedEvent.class);
            case "InventoryRestoration":
                return objectMapper.readValue(eventData,
InventoryRestorationEvent.class);
            case "BulkOrderProcessingCompleted":
                return objectMapper.readValue(eventData,
BulkOrderProcessingCompletedEvent.class);
            default:
                // Generic deserialization for unknown event types
                return objectMapper.readValue(eventData, Map.class);
        }
    }

    private void addEventHeaders(ProducerRecord<String, Object> producerRecord,
OutboxEvent event) {
        Headers headers = producerRecord.headers();

        // Add traceability headers
        headers.add("event-id", event.getId().toString().getBytes());
        headers.add("event-type", event.getEventType().getBytes());
        headers.add("aggregate-id", event.getAggregateId().getBytes());
        headers.add("aggregate-type", event.getAggregateType().getBytes());
        headers.add("created-at", event.getCreatedAt().toString().getBytes());
        headers.add("processing-attempt",
```

```
String.valueOf(event.getProcessingAttempts()).getBytes());

        // Add correlation headers for tracing
        headers.add("correlation-id", UUID.randomUUID().toString().getBytes());
        headers.add("source", "outbox-processor".getBytes());
    }

    private void handleFailedEvent(OutboxEvent event, Exception exception) {

        log.error("Handling permanently failed outbox event: eventId={}",
event.getId());

        try {
            // Create failure notification
            OutboxFailureNotification failureNotification =
OutboxFailureNotification.builder()
                .eventId(event.getId())
                .aggregateId(event.getAggregateId())
                .eventType(event.getEventType())
                .failureReason(exception.getMessage())
                .attempts(event.getProcessingAttempts())
                .failedAt(Instant.now())
                .build();

            // Send to dead letter queue or monitoring system
            kafkaTemplate.send("outbox-failures", event.getId().toString(),
failureNotification);

            meterRegistry.counter("outbox.events.permanently.failed",
                "event-type", event.getEventType()).increment();

        } catch (Exception e) {
            log.error("Failed to send failure notification for eventId={}",
event.getId(), e);
        }
    }

    private long calculateRetryDelay(int attemptNumber) {
        // Exponential backoff: 2^attempt * 1000ms
        return (long) (Math.pow(2, attemptNumber) * 1000);
    }
}

/**
 * Outbox processor configuration
 */
@Component
@ConfigurationProperties(prefix = "outbox.processor")
@lombok.Data
public class OutboxProcessorConfig {

    private long processingIntervalMs = 10000; // 10 seconds
    private long retryIntervalMs = 30000; // 30 seconds
    private long healthCheckIntervalMs = 60000; // 1 minute
```

```
    private String cleanupCron = "0 0 2 * * *"; // 2 AM daily
    private int batchSize = 100;
    private int maxRetryAttempts = 5;
    private int retentionDays = 7;
}

/**
 * Event data classes for outbox pattern
 */
@lombok.Data
@lombok.Builder
@lombok.NoArgsConstructor
@lombok.AllArgsConstructor
class OrderCreatedEvent {
    private String orderId;
    private String customerId;
    private String productId;
    private Integer quantity;
    private BigDecimal totalAmount;
    private Instant createdAt;
}

@lombok.Data
@lombok.Builder
@lombok.NoArgsConstructor
@lombok.AllArgsConstructor
class OrderStatusChangedEvent {
    private String orderId;
    private String previousStatus;
    private String newStatus;
    private String reason;
    private Instant changedAt;
}

@lombok.Data
@lombok.Builder
@lombok.NoArgsConstructor
@lombok.AllArgsConstructor
class PaymentConfirmedEvent {
    private String orderId;
    private BigDecimal amount;
    private Instant confirmedAt;
}

@lombok.Data
@lombok.Builder
@lombok.NoArgsConstructor
@lombok.AllArgsConstructor
class ShippingRequestedEvent {
    private String orderId;
    private String customerId;
    private String productId;
    private Integer quantity;
    private Instant requestedAt;
```

```
}

@lombok.Data
@lombok.Builder
@lombok.NoArgsConstructor
@lombok.AllArgsConstructor
class RefundRequestedEvent {
    private String orderId;
    private BigDecimal amount;
    private String reason;
    private Instant requestedAt;
}

@lombok.Data
@lombok.Builder
@lombok.NoArgsConstructor
@lombok.AllArgsConstructor
class InventoryRestorationEvent {
    private String orderId;
    private String productId;
    private Integer quantity;
    private String reason;
    private Instant requestedAt;
}

@lombok.Data
@lombok.Builder
@lombok.NoArgsConstructor
@lombok.AllArgsConstructor
class BulkOrderProcessingCompletedEvent {
    private String batchId;
    private Integer orderCount;
    private List<String> orderIds;
    private Instant completedAt;
}

@lombok.Data
@lombok.Builder
@lombok.NoArgsConstructor
@lombok.AllArgsConstructor
class OutboxFailureNotification {
    private UUID eventId;
    private String aggregateId;
    private String eventType;
    private String failureReason;
    private Integer attempts;
    private Instant failedAt;
}
```

📊 Comparisons & Trade-offs

Transaction Approaches Comparison

Approach	Consistency	Performance	Complexity	Use Case
At-Most-Once	❌ Low	★★★★★ Highest	★★★★★ Simple	Logging, Analytics
At-Least-Once	★★★ Medium	★★★★ High	★★★ Medium	Most Applications
Exactly-Once	★★★★★ Perfect	★★★ Medium	★★ Complex	Financial, Critical
Outbox Pattern	★★★★★ Perfect	★★ Lower	★ Most Complex	Microservices

Performance Impact Analysis

Configuration	Throughput (msg/sec)	Latency (ms)	Resource Usage
Non-Transactional	100,000	1	Low
Transactional EOS	15,000	10	Medium
Outbox Pattern	8,000	25	High

Transaction Coordinator Load

```
Transaction Coordinator Resource Usage:

Non-Transactional Producer:
CPU: ▓▒▒▒▒ 20%     Memory: ▓▒▒▒▒ 100MB     Network: ▓▒▒▒▒ Low

Transactional Producer (Low Volume):
CPU: ▓▓▒▒▒ 40%     Memory: ▓▓▒▒▒ 200MB     Network: ▓▓▒▒▒ Medium

Transactional Producer (High Volume):
CPU: ▓▓▓▓▒ 80%     Memory: ▓▓▓▒▒ 500MB     Network: ▓▓▓▓▒ High

Outbox Pattern (With Event Processor):
CPU: ▓▓▓▒▒ 60%     Memory: ▓▓▓▓▒ 800MB     Network: ▓▓▓▒▒ Medium

Key Observations:
- Transaction coordinator becomes bottleneck at high volumes
- Memory usage increases with transaction state tracking
- Network overhead from transaction coordination protocol
- Outbox pattern shifts load to database and event processor
```

🚨 Common Pitfalls & Best Practices

Critical Anti-Patterns to Avoid

❌ Transaction Configuration Mistakes

```
// DON'T - Missing or incorrect transactional.id configuration
@Bean
public ProducerFactory<String, Object> badTransactionalProducerFactory() {
    Map<String, Object> props = new HashMap<>();
    props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
    // BAD: Missing TRANSACTIONAL_ID_CONFIG
    // BAD: This won't enable transactions!
    props.put(ProducerConfig.ENABLE_IDEMPOTENCE_CONFIG, true);
    return new DefaultKafkaProducerFactory<>(props);
}

// DON'T - Non-unique transactional.id across instances
@Bean
public ProducerFactory<String, Object> badNonUniqueTransactionalId() {
    Map<String, Object> props = new HashMap<>();
    props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
    // BAD: Same transactional.id used by multiple instances
    props.put(ProducerConfig.TRANSACTIONAL_ID_CONFIG, "fixed-id");
    // This will cause zombie producer fencing issues
    return new DefaultKafkaProducerFactory<>(props);
}

// DON'T - Wrong isolation level for transactional consumers
@Bean
public ConsumerFactory<String, Object> badConsumerIsolation() {
    Map<String, Object> props = new HashMap<>();
    props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
    // BAD: Default isolation level won't see transactional guarantees
    // props.put(ConsumerConfig.ISOLATION_LEVEL_CONFIG, "read_committed");
    return new DefaultKafkaConsumerFactory<>(props);
}
```

❌ Error Handling Anti-Patterns

```
// DON'T - Catching transaction exceptions incorrectly
@Service
public class BadTransactionalService {

    @KafkaListener(topics = "orders")
    public void badErrorHandling(@Payload OrderMessage order) {
        try {
            processOrder(order);
            kafkaTemplate.send("processed-orders", order);
        } catch (Exception e) {
            // BAD: Swallowing exceptions prevents transaction rollback
```

```
            log.error("Error processing order", e);
            // Transaction will commit instead of rolling back
        }
    }

    @Transactional("kafkaTransactionManager")
    public void badRetryLogic(OrderMessage order) {
        try {
            processOrder(order);
        } catch (ProducerFencedException e) {
            // BAD: Trying to retry a non-recoverable exception
            retryProcessing(order); // This will always fail
        }
    }
}
```

❌ Outbox Pattern Anti-Patterns

```
// DON'T - Missing transactional boundary in outbox pattern
@Service
public class BadOutboxService {

    public void badOutboxImplementation(CreateOrderRequest request) {
        // BAD: No transaction boundary - dual write problem still exists
        Order order = orderRepository.save(createOrder(request));

        // If this fails, order is saved but event is not
        OutboxEvent event = createOutboxEvent(order);
        outboxEventRepository.save(event);
    }

    // DON'T - Processing outbox events without proper concurrency control
    @Scheduled(fixedDelay = 1000)
    public void badOutboxProcessor() {
        List<OutboxEvent> events = outboxEventRepository.findPendingEvents();

        // BAD: No concurrency control - multiple instances will process same
events
        events.forEach(event -> {
            kafkaTemplate.send(event.getTopic(), event.getData());
            event.setStatus(OutboxEventStatus.PROCESSED);
            outboxEventRepository.save(event);
        });
    }
}
```

Production Best Practices

☑ Optimal Transaction Configuration

```
/**
 * ☑ GOOD - Production-ready transactional configuration
 */
@Configuration
@lombok.extern.slf4j.Slf4j
public class ProductionTransactionalConfiguration {

    @Value("${spring.application.name:default-app}")
    private String applicationName;

    @Value("${server.port:8080}")
    private String serverPort;

    @Bean
    public ProducerFactory<String, Object>
productionTransactionalProducerFactory() {
        Map<String, Object> props = new HashMap<>();

        props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "${kafka.bootstrap-
servers}");
        props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
StringSerializer.class);
        props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
JsonSerializer.class);

        // GOOD: Unique transactional.id per application instance
        String transactionalId = applicationName + "-" + serverPort + "-" +
UUID.randomUUID().toString();
        props.put(ProducerConfig.TRANSACTIONAL_ID_CONFIG, transactionalId);

        // GOOD: Optimized transaction timeout
        props.put(ProducerConfig.TRANSACTION_TIMEOUT_CONFIG, 30000); // 30 seconds

        // GOOD: Production-optimized settings
        props.put(ProducerConfig.ACKS_CONFIG, "all");
        props.put(ProducerConfig.RETRIES_CONFIG, Integer.MAX_VALUE);
        props.put(ProducerConfig.BATCH_SIZE_CONFIG, 65536);
        props.put(ProducerConfig.LINGER_MS_CONFIG, 10);
        props.put(ProducerConfig.COMPRESSION_TYPE_CONFIG, "snappy");

        // GOOD: Connection pooling optimization
        props.put(ProducerConfig.CONNECTIONS_MAX_IDLE_MS_CONFIG, 600000);

        log.info("Configured production transactional producer: transactionalId=
{}", transactionalId);

        return new DefaultKafkaProducerFactory<>(props);
    }

    @Bean
    public ConsumerFactory<String, Object>
productionTransactionalConsumerFactory() {
        Map<String, Object> props = new HashMap<>();
```

```
        props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "${kafka.bootstrap-
servers}");
        props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class);
        props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
JsonDeserializer.class);
        props.put(ConsumerConfig.GROUP_ID_CONFIG, "${kafka.consumer.group-id}");

        // GOOD: Essential for exactly-once semantics
        props.put(ConsumerConfig.ISOLATION_LEVEL_CONFIG, "read_committed");
        props.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, false);

        // GOOD: Production-optimized consumer settings
        props.put(ConsumerConfig.SESSION_TIMEOUT_MS_CONFIG, 30000);
        props.put(ConsumerConfig.HEARTBEAT_INTERVAL_MS_CONFIG, 10000);
        props.put(ConsumerConfig.MAX_POLL_INTERVAL_MS_CONFIG, 300000);
        props.put(ConsumerConfig.MAX_POLL_RECORDS_CONFIG, 100); // Smaller batches
for EOS

        return new DefaultKafkaConsumerFactory<>(props);
    }
}

/**
 * ☑ GOOD - Robust error handling for transactions
 */
@Service
@lombok.extern.slf4j.Slf4j
public class RobustTransactionalService {

    @Autowired
    private KafkaTemplate<String, Object> kafkaTemplate;

    @Autowired
    private MeterRegistry meterRegistry;

    @KafkaListener(
        topics = "orders",
        containerFactory = "transactionalListenerContainerFactory"
    )
    public void processWithRobustErrorHandling(@Payload OrderMessage order,
                                               @Header(KafkaHeaders.OFFSET) long
offset) {

        Timer.Sample sample = Timer.start(meterRegistry);

        try {
            log.info("Processing order transactionally: orderId={}, offset={}",
order.getOrderId(), offset);

            // Business logic processing
            ProcessedOrder processedOrder = processOrderSafely(order);
```

```
            // Send to output topic within same transaction
            kafkaTemplate.send("processed-orders", order.getOrderId(),
processedOrder);

            // Update success metrics
            meterRegistry.counter("orders.processed.success").increment();

            log.info("Order processed successfully: orderId={}",
order.getOrderId());

        } catch (NonRetryableException e) {
            // Send to DLQ for manual inspection
            sendToDLQ(order, e);
            meterRegistry.counter("orders.processed.dlq").increment();

            log.error("Order sent to DLQ: orderId={}", order.getOrderId(), e);

        } catch (RetryableException e) {
            // Let transaction rollback for automatic retry
            meterRegistry.counter("orders.processed.retry").increment();

            log.warn("Order processing will be retried: orderId={}",
order.getOrderId(), e);
            throw e;

        } catch (ProducerFencedException | FencedInstanceIdException |
OutOfOrderSequenceException e) {
            // Fatal exceptions - requires application restart
            meterRegistry.counter("orders.processed.fatal").increment();

            log.error("Fatal transaction error - application should restart:
orderId={}", order.getOrderId(), e);

            // Gracefully shutdown application
            initiateGracefulShutdown();

            throw e;

        } catch (Exception e) {
            // Unexpected errors
            meterRegistry.counter("orders.processed.error").increment();

            log.error("Unexpected error processing order: orderId={}",
order.getOrderId(), e);
            throw e;

        } finally {
            sample.stop(Timer.builder("orders.processing.duration")
                .tag("result", "completed")
                .register(meterRegistry));
        }
    }

    /**
```

```
     * ☑ GOOD - Graceful handling of processing errors
     */
    private ProcessedOrder processOrderSafely(OrderMessage order) {

        try {
            // Validate order
            validateOrder(order);

            // Process business logic
            return executeBusinessLogic(order);

        } catch (ValidationException e) {
            // Validation errors are not retryable
            throw new NonRetryableException("Order validation failed", e);

        } catch (ExternalServiceException e) {
            // External service errors may be retryable
            if (e.isRetryable()) {
                throw new RetryableException("External service temporarily
unavailable", e);
            } else {
                throw new NonRetryableException("External service permanently
failed", e);
            }

        } catch (Exception e) {
            // Unknown errors are retryable by default
            throw new RetryableException("Unknown processing error", e);
        }
    }

    private void sendToDLQ(OrderMessage order, Exception exception) {
        try {
            DLQMessage dlqMessage = DLQMessage.builder()
                .originalMessage(order)
                .errorMessage(exception.getMessage())
                .errorClass(exception.getClass().getName())
                .timestamp(Instant.now())
                .build();

            kafkaTemplate.send("orders-dlq", order.getOrderId(), dlqMessage);

        } catch (Exception e) {
            log.error("Failed to send message to DLQ: orderId={}",
order.getOrderId(), e);
        }
    }

    private void validateOrder(OrderMessage order) {
        if (order.getOrderId() == null || order.getOrderId().isEmpty()) {
            throw new ValidationException("Order ID is required");
        }
        if (order.getAmount() == null ||
order.getAmount().compareTo(BigDecimal.ZERO) <= 0) {
```

```
            throw new ValidationException("Order amount must be positive");
        }
    }

    private ProcessedOrder executeBusinessLogic(OrderMessage order) {
        // Business logic implementation
        return ProcessedOrder.builder()
            .orderId(order.getOrderId())
            .status("PROCESSED")
            .processedAt(Instant.now())
            .build();
    }

    private void initiateGracefulShutdown() {
        // Implementation would trigger application shutdown
        log.error("Initiating graceful application shutdown due to fatal Kafka
error");
    }
}
```

🛠️ CLI Commands and Monitoring

Kafka CLI Commands for Transaction Monitoring

```
#!/bin/bash

# Monitor transaction coordinator
kafka-log-dirs.sh \
  --bootstrap-server localhost:9092 \
  --topic-list __transaction_state \
  --describe

# Check transaction state topic
kafka-topics.sh \
  --bootstrap-server localhost:9092 \
  --describe \
  --topic __transaction_state

# Monitor consumer group with exactly-once semantics
kafka-consumer-groups.sh \
  --bootstrap-server localhost:9092 \
  --group exactly-once-group \
  --describe

# Check producer metrics for transactional producer
kafka-run-class.sh kafka.tools.JmxTool \
  --object-name kafka.producer:type=producer-metrics,client-id=transactional-
producer \
  --jmx-url service:jmx:rmi:///jndi/rmi://localhost:9999/jmxrmi
```

```
# Monitor transaction coordinator metrics
kafka-run-class.sh kafka.tools.JmxTool \
  --object-name kafka.coordinator.transaction:type=TransactionCoordinator \
  --jmx-url service:jmx:rmi:///jndi/rmi://localhost:9999/jmxrmi

# Test transactional producer
kafka-console-producer.sh \
  --bootstrap-server localhost:9092 \
  --topic test-transactions \
  --producer-property transactional.id=test-tx-1

# Test read_committed consumer
kafka-console-consumer.sh \
  --bootstrap-server localhost:9092 \
  --topic test-transactions \
  --from-beginning \
  --isolation-level read_committed

# Monitor outbox table (PostgreSQL)
psql -h localhost -d myapp -c "
SELECT
    status,
    COUNT(*) as count,
    MIN(created_at) as oldest,
    MAX(created_at) as newest
FROM outbox_events
GROUP BY status;
"
```

JMX Metrics for Transaction Monitoring

```
# Key transaction metrics to monitor

# Producer transaction metrics
kafka.producer.transaction-send-rate: Rate of transactional sends
kafka.producer.transaction-duration-avg: Average transaction duration
kafka.producer.transaction-duration-max: Maximum transaction duration

# Consumer transaction metrics
kafka.consumer.transaction-read-rate: Rate of transactional reads
kafka.consumer.committed-time-ns-total: Time spent committing

# Coordinator metrics
kafka.coordinator.transaction.transaction-abort-rate: Transaction abort rate
kafka.coordinator.transaction.transaction-commit-rate: Transaction commit rate
kafka.coordinator.transaction.active-transactions: Current active transactions

# Application metrics (custom)
outbox.events.pending.count: Pending outbox events
outbox.events.processing.duration: Processing time per event
```

```
  orders.processed.success: Successfully processed orders
  orders.processed.error: Failed order processing
```

Version Highlights

Spring Kafka Transaction Evolution

Version	Release	Key Transaction Features
3.2.x	2024	**Enhanced EOS performance**, batch transaction improvements
3.1.x	2024	**KRaft transaction coordinator** support, observability enhancements
3.0.x	2023	**EOSMode.V2 only**, native compilation support
2.9.x	2022	**KafkaAwareTransactionManager** improvements, batch EOS
2.8.x	2022	**Enhanced exactly-once semantics**, transaction error handling
2.7.x	2021	**@Transactional improvements**, ChainedKafkaTransactionManager
2.6.x	2021	**KIP-447 fetch-offset-request fencing** (EOSMode.V2)
2.5.x	2020	**EOSMode.V2 introduction**, improved transaction coordinator
2.4.x	2020	**KafkaAwareTransactionManager**, better JTA integration
2.3.x	2019	**Initial transaction support**, basic exactly-once semantics

Critical Apache Kafka Version Dependencies

Kafka 3.5+ (2023):

- **KRaft Transaction Coordinator**: Native ZooKeeper-free transaction coordination
- **Enhanced Performance**: Improved transaction throughput and latency
- **Better Observability**: Enhanced metrics and monitoring capabilities

Kafka 2.8+ (2021):

- **KRaft Mode Support**: Transition away from ZooKeeper dependency
- **Transaction Performance**: Significant improvements in transaction processing

Kafka 2.5+ (2020):

- **EOSMode.V2**: Fetch-offset-request fencing eliminates need for multiple producers
- **Producer Scalability**: Single producer can handle multiple input partitions
- **Performance Improvement**: Reduced overhead in exactly-once processing

Spring Boot Compatibility Matrix:

Spring Boot	Spring Kafka	Min Kafka Broker	EOS Support
3.2.x	3.1.x	2.8+	☑ V2 Only

Spring Boot	Spring Kafka	Min Kafka Broker	EOS Support
3.1.x	3.0.x	2.8+	☑ V2 Only
3.0.x	2.9.x	2.8+	☑ V2 Only
2.7.x	2.8.x	2.5+	☑ V2 Only
2.6.x	2.7.x	2.5+	☑ V1 + V2

Production Checklist

Essential Configuration Checklist

- ☑ **Unique transactional.id** per application instance
- ☑ **isolation.level=read_committed** for transactional consumers
- ☑ **enable.auto.commit=false** for exactly-once consumers
- ☑ **Appropriate transaction.timeout** configuration
- ☑ **KafkaAwareTransactionManager** for consumer offset management
- ☑ **Proper error handling** for ProducerFencedException
- ☑ **Dead Letter Topic** strategy for non-retryable failures
- ☑ **Monitoring and alerting** for transaction metrics
- ☑ **Health checks** for transaction coordinator connectivity
- ☑ **Graceful shutdown** handling for transactional applications

Outbox Pattern Checklist

- ☑ **Single database transaction** for business logic + outbox event
- ☑ **Idempotent event processor** with proper concurrency control
- ☑ **Retry logic** with exponential backoff for failed events
- ☑ **Dead letter handling** for permanently failed events
- ☑ **Event cleanup** strategy to prevent table growth
- ☑ **Monitoring** for outbox table health and processing metrics
- ☑ **Backup and recovery** procedures for outbox events

Key Takeaways

When to Use Each Approach

1. **Producer Transactions**: Single-service, multi-topic publishing with exactly-once guarantees
2. **Consumer Transactions**: Read-process-write patterns requiring exactly-once semantics
3. **Outbox Pattern**: Cross-service consistency with database + Kafka atomicity
4. **Chain Transactions**: Complex multi-step workflows with rollback requirements

Performance Considerations

- **Exactly-Once Semantics**: 5-10x throughput reduction vs at-least-once
- **Transaction Coordinator**: Can become bottleneck at high volumes
- **Outbox Pattern**: Additional database load + eventual consistency delay

- **Memory Usage**: Increased with transaction state tracking

Critical Success Factors

1. **Unique Transaction IDs**: Prevent zombie producer issues
2. **Proper Error Handling**: Distinguish retryable vs fatal exceptions
3. **Monitoring**: Track transaction metrics and failures
4. **Testing**: Validate failure scenarios and recovery procedures
5. **Capacity Planning**: Account for transaction coordinator overhead

Last Updated: September 2025
Spring Kafka Version Coverage: 3.2.x
Apache Kafka Compatibility: 3.5.x
Spring Boot Version: 3.2.x

This comprehensive guide provides production-ready patterns for implementing exactly-once semantics in Spring Kafka applications, from basic transactions to advanced outbox patterns, ensuring data consistency and reliability in distributed systems.

[613] [614] [615]