Spring Kafka Transactions: Part 2 - Consumer Offsets & Outbox Pattern

Continuation of the comprehensive guide covering consumer offsets within transactions, database transaction management, and the transactional outbox pattern with production examples.

📍 Consumer Offsets within Transactions

Simple Explanation: Consumer offsets within transactions ensure that offset commits are part of the same atomic operation as message processing and publishing. This prevents message loss or duplication by ensuring that consumer position is only updated when the entire transaction succeeds.

Why Consumer Offset Management in Transactions Matters:

- **Exactly-Once Processing**: Prevents reprocessing of successfully handled messages
- **Consistency Guarantee**: Offset commits aligned with business logic completion
- **Failure Recovery**: Proper rollback ensures messages are reprocessed after failures
- **Duplicate Prevention**: Consumer won't advance past uncommitted transactions
- **Data Integrity**: Maintains consistency between consumed and produced messages

Consumer Offset Transaction Flow:

```
Consumer Offset Transaction Management:

┌─────────────────────────────────────────────────────────────┐
│                   Input Topic (orders)                      │
│  [Order1][Order2][Order3][Order4][Order5]                 │
│           ^                                                 │
│           │ Current Consumer Offset: 2                     │
└─────────────────────────────────────────────────────────────┘
                              │
                              │ Read Order3 (offset 3)
                              ▼
┌─────────────────────────────────────────────────────────────┐
│                 Transactional Consumer                      │
│  ┌──────────────────────────────────────────────────────────┤
│  │ 1. beginTransaction()                                  │
│  │ 2. Process Order3 (business logic)                    │
│  │ 3. Send to output topics (if any)                     │
│  │ 4. sendOffsetsToTransaction(offset: 3)              │
│  │ 5. commitTransaction()                                │
│  └──────────────────────────────────────────────────────────┤
│  All operations atomic - offset commits only if success     │
└─────────────────────────────────────────────────────────────┘
                              │
                              │ On Success: Offset advances to 3
                              │ On Failure: Offset remains at 2, Order3 reprocessed
                              ▼
┌─────────────────────────────────────────────────────────────┐
│                    Output Topic(s)                          │
│  [ProcessedOrder1][ProcessedOrder2][ProcessedOrder3]       │
│  Only visible to read_committed consumers                   │
└─────────────────────────────────────────────────────────────┘

Exactly-Once Consumer-Producer Pattern:
Input → [READ] → [PROCESS] → [WRITE] → [COMMIT OFFSET] → Output
                   ↑                                       ↑
              All operations within single transaction boundary
```

Advanced Consumer Offset Transaction Configuration

```java
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.kafka.annotation.EnableKafka;
import org.springframework.kafka.config.ConcurrentKafkaListenerContainerFactory;
import org.springframework.kafka.core.ConsumerFactory;
import org.springframework.kafka.core.DefaultKafkaConsumerFactory;
import org.springframework.kafka.listener.ContainerProperties;
import org.springframework.kafka.transaction.KafkaAwareTransactionManager;

import org.apache.kafka.clients.consumer.ConsumerConfig;
import org.apache.kafka.common.serialization.StringDeserializer;

/**
 * Transactional consumer configuration with offset management
 */
@Configuration
@EnableKafka
@lombok.extern.slf4j.Slf4j
public class TransactionalConsumerConfiguration {

    @Autowired
    private ProducerFactory<String, Object> transactionalProducerFactory;

    /**
     * Transactional consumer factory for exactly-once processing
     */
    @Bean
    public ConsumerFactory<String, Object> transactionalConsumerFactory() {
        Map<String, Object> props = new HashMap<>();

        // Basic consumer configuration
        props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        props.put(ConsumerConfig.GROUP_ID_CONFIG, "transactional-consumer-group");
        props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class);
        props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
JsonDeserializer.class);

        // CRITICAL: Enable read_committed for transactional consumers
        props.put(ConsumerConfig.ISOLATION_LEVEL_CONFIG, "read_committed");
        // Disable auto commit - managed by transaction
        props.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, false);

        // Optimize for exactly-once processing
        props.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest");
        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); // 5
minutes

        // Set lower fetch sizes for transactional processing
        props.put(ConsumerConfig.FETCH_MIN_BYTES_CONFIG, 1024); // 1KB
        props.put(ConsumerConfig.FETCH_MAX_WAIT_MS_CONFIG, 500);
        props.put(ConsumerConfig.MAX_POLL_RECORDS_CONFIG, 100); // Smaller batches

        return new DefaultKafkaConsumerFactory<>(props);
    }

    /**
     * Transactional listener container factory with KafkaAwareTransactionManager
     */
    @Bean
    public ConcurrentKafkaListenerContainerFactory<String, Object>
transactionalListenerContainerFactory() {
        ConcurrentKafkaListenerContainerFactory<String, Object> factory =
            new ConcurrentKafkaListenerContainerFactory<>();

        factory.setConsumerFactory(transactionalConsumerFactory());

        // CRITICAL: Set KafkaAwareTransactionManager for exactly-once semantics

factory.getContainerProperties().setTransactionManager(kafkaAwareTransactionManage
r());

        // Container properties for transactional processing
        ContainerProperties containerProps = factory.getContainerProperties();
        containerProps.setAckMode(ContainerProperties.AckMode.RECORD);
        containerProps.setSyncCommits(true); // Synchronous commits for safety
        containerProps.setDeliveryAttemptHeader(true); // Track delivery attempts

        // Configure for single-threaded processing to maintain order
        factory.setConcurrency(1); // Single consumer per partition for EOS

        log.info("Configured transactional listener container factory with
exactly-once semantics");

        return factory;
    }

    /**
     * Kafka-aware transaction manager for consumer offset management
     */
    @Bean
    public KafkaAwareTransactionManager kafkaAwareTransactionManager() {
        return new KafkaAwareTransactionManager(transactionalProducerFactory);
    }

    /**
     * High-throughput transactional container factory (multiple consumers)
     */
    @Bean("highThroughputTransactionalFactory")
    public ConcurrentKafkaListenerContainerFactory<String, Object>
highThroughputTransactionalFactory() {
        ConcurrentKafkaListenerContainerFactory<String, Object> factory =
            new ConcurrentKafkaListenerContainerFactory<>();

        factory.setConsumerFactory(transactionalConsumerFactory());

factory.getContainerProperties().setTransactionManager(kafkaAwareTransactionManage
r());

        // Higher concurrency for throughput (trades off ordering within
partition)
        factory.setConcurrency(3);

        // Optimize for throughput

factory.getContainerProperties().setAckMode(ContainerProperties.AckMode.BATCH);
        factory.getContainerProperties().setPollTimeout(Duration.ofMillis(1000));

        log.info("Configured high-throughput transactional container factory:
concurrency=3");

        return factory;
    }
}

/**
 * Exactly-once consumer service with offset transaction management
 */
@Service
@lombok.extern.slf4j.Slf4j
public class ExactlyOnceConsumerService {

    @Autowired
    private KafkaTemplate<String, Object> transactionalKafkaTemplate;

    @Autowired
    private OrderProcessingService orderProcessingService;

    @Autowired
    private PaymentProcessingService paymentProcessingService;

    /**
     * Exactly-once order processing with automatic offset management
     */
    @KafkaListener(
        topics = "raw-orders",
        groupId = "exactly-once-order-processor",
        containerFactory = "transactionalListenerContainerFactory"
    )
    public void processOrderExactlyOnce(@Payload OrderMessage orderMessage,
                                        @Header(KafkaHeaders.RECEIVED_TOPIC) String
topic,
                                        @Header(KafkaHeaders.RECEIVED_PARTITION) int
partition,
                                        @Header(KafkaHeaders.OFFSET) long offset) {

        String orderId = orderMessage.getOrderId();

        log.info("Processing order exactly-once: orderId={}, topic={}, partition=
{}, offset={}",
            orderId, topic, partition, offset);

        try {
            // The @KafkaListener with transactional container automatically:
            // 1. Begins transaction
            // 2. Processes this method
            // 3. Commits consumer offset within transaction
            // 4. Commits transaction (or rolls back on exception)

            // Step 1: Validate order
            ValidationResult validation =
orderProcessingService.validateOrder(orderMessage);
            if (!validation.isValid()) {
                throw new OrderValidationException("Order validation failed: " +
validation.getReason());
            }

            // Step 2: Process order business logic
            ProcessedOrder processedOrder =
orderProcessingService.processOrder(orderMessage);

            // Step 3: Send processed order to output topic (within same
transaction)
            transactionalKafkaTemplate.send("processed-orders", orderId,
processedOrder);

            // Step 4: Send additional events as needed
            if (processedOrder.requiresPayment()) {
                PaymentRequest paymentRequest =
createPaymentRequest(processedOrder);
                transactionalKafkaTemplate.send("payment-requests",
paymentRequest.getPaymentId(), paymentRequest);
            }

            if (processedOrder.requiresShipping()) {
                ShippingRequest shippingRequest =
createShippingRequest(processedOrder);
                transactionalKafkaTemplate.send("shipping-requests",
shippingRequest.getShippingId(), shippingRequest);
```

```
            }

            // Step 5: Send audit event
            AuditEvent auditEvent = AuditEvent.builder()
                .auditId(UUID.randomUUID().toString())
                .eventType("ORDER_PROCESSED")
                .orderId(orderId)
                .timestamp(Instant.now())
                .details(Map.of(
                    "topic", topic,
                    "partition", partition,
                    "offset", offset,
                    "processingTime", System.currentTimeMillis()
                ))
                .build();

            transactionalKafkaTemplate.send("audit-events",
auditEvent.getAuditId(), auditEvent);

            log.info("Order processed successfully with exactly-once semantics:
orderId={}, offset={}",
                orderId, offset);

            // Transaction commits automatically here, including consumer offset

        } catch (Exception e) {
            log.error("Error processing order exactly-once: orderId={}, offset=
{}", orderId, offset, e);

            // Exception triggers automatic transaction rollback
            // Consumer offset will NOT advance - message will be reprocessed
            throw e;
        }
    }

    /**
     * Exactly-once payment processing with conditional output
     */
    @KafkaListener(
        topics = "payment-requests",
        groupId = "exactly-once-payment-processor",
        containerFactory = "transactionalListenerContainerFactory"
    )
    public void processPaymentExactlyOnce(@Payload PaymentRequest paymentRequest,
                                          @Header(KafkaHeaders.OFFSET) long offset)
{

        String paymentId = paymentRequest.getPaymentId();

        log.info("Processing payment exactly-once: paymentId={}, offset={}",
paymentId, offset);

        try {
            // Step 1: Process payment
            PaymentResult paymentResult =
paymentProcessingService.processPayment(paymentRequest);

            // Step 2: Send result based on payment outcome
            if (paymentResult.isSuccessful()) {
                // Success flow
                transactionalKafkaTemplate.send("payment-confirmations",
paymentId,
                    createPaymentConfirmation(paymentResult));

                transactionalKafkaTemplate.send("order-status-updates",
paymentResult.getOrderId(),
                    createOrderStatusUpdate(paymentResult.getOrderId(), "PAID"));

                transactionalKafkaTemplate.send("accounting-entries",
UUID.randomUUID().toString(),
                    createAccountingEntry(paymentResult));

                log.info("Payment processed successfully: paymentId={}",
paymentId);

            } else {
                // Failure flow
                transactionalKafkaTemplate.send("payment-failures", paymentId,
                    createPaymentFailure(paymentResult));

                transactionalKafkaTemplate.send("order-status-updates",
paymentResult.getOrderId(),
                    createOrderStatusUpdate(paymentResult.getOrderId(),
"PAYMENT_FAILED"));

                // Schedule retry if retryable
                if (paymentResult.isRetryable()) {
                    PaymentRetrySchedule retrySchedule =
createPaymentRetrySchedule(paymentRequest);
                    transactionalKafkaTemplate.send("payment-retry-schedule",
                        retrySchedule.getScheduleId(), retrySchedule);
                }

                log.warn("Payment failed: paymentId={}, reason={}", paymentId,
paymentResult.getFailureReason());
            }

            // Step 3: Always send audit event
            AuditEvent auditEvent = AuditEvent.builder()
                .auditId(UUID.randomUUID().toString())
                .eventType("PAYMENT_PROCESSED")
                .paymentId(paymentId)
                .timestamp(Instant.now())
                .details(Map.of(
                    "success", paymentResult.isSuccessful(),
                    "amount", paymentResult.getAmount().toString(),
                    "offset", offset
                ))
                .build();

            transactionalKafkaTemplate.send("audit-events",
auditEvent.getAuditId(), auditEvent);

            // Consumer offset commits within transaction automatically

        } catch (Exception e) {
            log.error("Error processing payment exactly-once: paymentId={},
offset={}", paymentId, offset, e);
            throw e; // Triggers transaction rollback and reprocessing
        }
    }

    /**
     * Complex exactly-once processing with multiple input sources
     */
    @KafkaListener(
        topics = {"order-updates", "inventory-updates", "customer-updates"},
        groupId = "multi-source-exactly-once-processor",
        containerFactory = "transactionalListenerContainerFactory"
    )
    public void processMultiSourceExactlyOnce(@Payload Object message,
                                              @Header(KafkaHeaders.RECEIVED_TOPIC)
String topic,

@Header(KafkaHeaders.RECEIVED_PARTITION) int partition,
                                              @Header(KafkaHeaders.OFFSET) long
offset) {

        log.info("Processing multi-source message exactly-once: topic={},
partition={}, offset={}",
            topic, partition, offset);

        try {
            // Route based on topic
            switch (topic) {
                case "order-updates" -> processOrderUpdate((OrderUpdate) message,
offset);
                case "inventory-updates" ->
processInventoryUpdate((InventoryUpdate) message, offset);
                case "customer-updates" -> processCustomerUpdate((CustomerUpdate)
message, offset);
                default -> throw new IllegalArgumentException("Unknown topic: " +
topic);
            }

            // Send cross-topic correlation event
            CrossTopicCorrelation correlation = CrossTopicCorrelation.builder()
                .correlationId(UUID.randomUUID().toString())
                .sourceTopic(topic)
                .sourcePartition(partition)
                .sourceOffset(offset)
                .processedAt(Instant.now())
                .build();

            transactionalKafkaTemplate.send("cross-topic-correlations",
                correlation.getCorrelationId(), correlation);

            log.info("Multi-source processing completed: topic={}, offset={}",
topic, offset);

        } catch (Exception e) {
            log.error("Error in multi-source exactly-once processing: topic={},
offset={}", topic, offset, e);
            throw e;
        }
    }

    /**
     * Batch exactly-once processing with offset management
     */
    @KafkaListener(
        topics = "bulk-operations",
        groupId = "batch-exactly-once-processor",
        containerFactory = "transactionalListenerContainerFactory"
    )
    public void processBatchExactlyOnce(@Payload List<BulkOperation>
bulkOperations,
                                      @Header(KafkaHeaders.RECEIVED_PARTITION)
List<Integer> partitions,
                                      @Header(KafkaHeaders.OFFSET) List<Long>
offsets) {

        log.info("Processing batch exactly-once: size={}, partitions={}, offsets=
{}",
            bulkOperations.size(), partitions, offsets);

        try {
            // Process each operation in the batch
            for (int i = 0; i < bulkOperations.size(); i++) {
                BulkOperation operation = bulkOperations.get(i);
                Long offset = offsets.get(i);
                Integer partition = partitions.get(i);

                log.debug("Processing bulk operation: index={}, partition={},
offset={}", i, partition, offset);

                // Process individual operation
                BulkOperationResult result = processBulkOperation(operation);

                // Send result
                transactionalKafkaTemplate.send("bulk-operation-results",
operation.getOperationId(), result);

                // Send audit for individual operation
                AuditEvent auditEvent = AuditEvent.builder()
                    .auditId(UUID.randomUUID().toString())
```

```
                        .eventType("BULK_OPERATION_PROCESSED")
                        .operationId(operation.getOperationId())
                        .timestamp(Instant.now())
                        .details(Map.of(
                            "partition", partition,
                            "offset", offset,
                            "success", result.isSuccessful()
                        ))
                        .build();

                    transactionalKafkaTemplate.send("audit-events",
auditEvent.getAuditId(), auditEvent);
                }

                // Send batch completion event
                BatchCompletionEvent batchCompletion = BatchCompletionEvent.builder()
                    .batchId(UUID.randomUUID().toString())
                    .operationCount(bulkOperations.size())
                    .processedAt(Instant.now())

.partitions(partitions.stream().distinct().collect(Collectors.toList()))
                    .offsetRanges(calculateOffsetRanges(partitions, offsets))
                    .build();

                transactionalKafkaTemplate.send("batch-completions",
batchCompletion.getBatchId(), batchCompletion);

                log.info("Batch exactly-once processing completed: size={}, batchId=
{}",
                    bulkOperations.size(), batchCompletion.getBatchId());

                // All consumer offsets commit automatically within transaction

            } catch (Exception e) {
                log.error("Error in batch exactly-once processing: size={}",
bulkOperations.size(), e);
                throw e; // All offsets will rollback, entire batch reprocessed
            }
        }

        // Helper methods for exactly-once processing
        private PaymentRequest createPaymentRequest(ProcessedOrder processedOrder) {
            return PaymentRequest.builder()
                .paymentId(UUID.randomUUID().toString())
                .orderId(processedOrder.getOrderId())
                .amount(processedOrder.getTotalAmount())
                .customerId(processedOrder.getCustomerId())
                .paymentMethod(processedOrder.getPaymentMethod())
                .build();
        }

        private ShippingRequest createShippingRequest(ProcessedOrder processedOrder) {
            return ShippingRequest.builder()
                .shippingId(UUID.randomUUID().toString())
                .orderId(processedOrder.getOrderId())
                .customerId(processedOrder.getCustomerId())
                .shippingAddress(processedOrder.getShippingAddress())
                .priority(determinePriority(processedOrder))
                .build();
    }

    private PaymentConfirmation createPaymentConfirmation(PaymentResult
paymentResult) {
        return PaymentConfirmation.builder()
                .paymentId(paymentResult.getPaymentId())
                .orderId(paymentResult.getOrderId())
                .amount(paymentResult.getAmount())
                .confirmationNumber(paymentResult.getConfirmationNumber())
                .confirmedAt(Instant.now())
                .build();
    }

    private OrderStatusUpdate createOrderStatusUpdate(String orderId, String
status) {
        return OrderStatusUpdate.builder()
                .orderId(orderId)
                .status(status)
                .updatedAt(Instant.now())
                .build();
    }

    private AccountingEntry createAccountingEntry(PaymentResult paymentResult) {
        return AccountingEntry.builder()
                .entryId(UUID.randomUUID().toString())
                .paymentId(paymentResult.getPaymentId())
                .amount(paymentResult.getAmount())
                .accountId(paymentResult.getAccountId())
                .entryType("PAYMENT_RECEIVED")
                .createdAt(Instant.now())
                .build();
    }

    private PaymentFailure createPaymentFailure(PaymentResult paymentResult) {
        return PaymentFailure.builder()
                .paymentId(paymentResult.getPaymentId())
                .orderId(paymentResult.getOrderId())
                .failureReason(paymentResult.getFailureReason())
                .isRetryable(paymentResult.isRetryable())
                .failedAt(Instant.now())
                .build();
    }

    private PaymentRetrySchedule createPaymentRetrySchedule(PaymentRequest
paymentRequest) {
        return PaymentRetrySchedule.builder()
                .scheduleId(UUID.randomUUID().toString())
                .paymentId(paymentRequest.getPaymentId())
                .originalRequest(paymentRequest)
```

```
            .retryAfter(Instant.now().plus(Duration.ofMinutes(15)))
            .maxRetries(3)
            .currentRetryCount(1)
            .build();
    }

    private void processOrderUpdate(OrderUpdate orderUpdate, long offset) {
        log.debug("Processing order update: orderId={}, offset={}",
orderUpdate.getOrderId(), offset);

        // Process order update logic
        OrderUpdateResult result =
orderProcessingService.updateOrder(orderUpdate);

        // Send update result
        transactionalKafkaTemplate.send("order-update-results",
orderUpdate.getOrderId(), result);
    }

    private void processInventoryUpdate(InventoryUpdate inventoryUpdate, long
offset) {
        log.debug("Processing inventory update: productId={}, offset={}",
inventoryUpdate.getProductId(), offset);

        // Process inventory update logic
        InventoryUpdateResult result =
orderProcessingService.updateInventory(inventoryUpdate);

        // Send update result
        transactionalKafkaTemplate.send("inventory-update-results",
inventoryUpdate.getProductId(), result);
    }

    private void processCustomerUpdate(CustomerUpdate customerUpdate, long offset)
{
        log.debug("Processing customer update: customerId={}, offset={}",
customerUpdate.getCustomerId(), offset);

        // Process customer update logic
        CustomerUpdateResult result =
orderProcessingService.updateCustomer(customerUpdate);

        // Send update result
        transactionalKafkaTemplate.send("customer-update-results",
customerUpdate.getCustomerId(), result);
    }

    private BulkOperationResult processBulkOperation(BulkOperation operation) {
        // Process individual bulk operation
        return BulkOperationResult.builder()
            .operationId(operation.getOperationId())
            .successful(true)
            .processedAt(Instant.now())
            .build();
    }

    private String determinePriority(ProcessedOrder processedOrder) {
        return processedOrder.getTotalAmount().compareTo(new BigDecimal("1000")) >
0 ?
            "HIGH" : "STANDARD";
    }

    private Map<Integer, String> calculateOffsetRanges(List<Integer> partitions,
List<Long> offsets) {
        Map<Integer, List<Long>> offsetsByPartition = IntStream.range(0,
partitions.size())
            .boxed()
            .collect(Collectors.groupingBy(
                partitions::get,
                Collectors.mapping(offsets::get, Collectors.toList())
            ));

        return offsetsByPartition.entrySet().stream()
            .collect(Collectors.toMap(
                Map.Entry::getKey,
                entry -> {
                    List<Long> partitionOffsets = entry.getValue();
                    long min = Collections.min(partitionOffsets);
                    long max = Collections.max(partitionOffsets);
                    return min == max ? String.valueOf(min) : min + "-" + max;
                }
            ));
    }
}
```

🏦 Database + Kafka Transaction Management (Outbox Pattern)

Simple Explanation: The Transactional Outbox Pattern solves the dual-write problem by ensuring that database operations and Kafka message publishing happen atomically. Instead of writing directly to Kafka, events are written to an "outbox" table in the same database transaction, then asynchronously published to Kafka by a separate process.

Why the Outbox Pattern is Essential:

- **Atomic Consistency**: Database and messaging operations are atomic
- **Dual-Write Problem Solution**: Eliminates the risk of partial failures
- **Reliability**: Guarantees at-least-once delivery to Kafka
- **Resilience**: Survives application crashes and restarts
- **Decoupling**: Separates business logic from messaging infrastructure

Outbox Pattern Architecture:

```
Traditional Dual-Write Problem:
┌──────────────────────────────────────────────────────────────┐
```

```
│                  Business Transaction                          │
│  1. Update Database      ✓ SUCCESS                            │
│  2. Send to Kafka        ✗ FAILURE                            │
│  Result: Database inconsistent with Kafka                     │
└──────────────────────────────────────────────────────────────┘

Outbox Pattern Solution:
┌──────────────────────────────────────────────────────────────┐
│             Single Database Transaction                        │
│  ┌─────────────────────────────────────────────────────────┤
│  │ 1. Update Business Data                                │
│  │ 2. Insert Event to Outbox Table                       │
│  │ Both operations succeed or fail together               │
│  └─────────────────────────────────────────────────────────┤
└──────────────────────────────────────────────────────────────┘
                      │
                      │
                      ▼ (Separate Process)
┌──────────────────────────────────────────────────────────────┐
│               Outbox Event Processor                           │
│  1. Poll Outbox Table for new events                          │
│  2. Publish events to Kafka                                   │
│  3. Mark events as processed                                  │
│  4. Delete/Archive processed events                           │
└──────────────────────────────────────────────────────────────┘
                      │
                      │
                      ▼
┌──────────────────────────────────────────────────────────────┐
│                     Kafka Topics                               │
│  Events guaranteed to be published (at least once)            │
└──────────────────────────────────────────────────────────────┘
```

Spring Boot Outbox Pattern Implementation

```java
/**
 * Outbox pattern entities and database schema
 */
@Entity
@Table(name = "outbox_events")
@lombok.Data
@lombok.Builder
@lombok.NoArgsConstructor
@lombok.AllArgsConstructor
public class OutboxEvent {

    @Id
    @GeneratedValue(strategy = GenerationType.UUID)
    @Column(name = "id", columnDefinition = "uuid")
    private UUID id;

    @Column(name = "aggregate_id", nullable = false)
    private String aggregateId;
    @Column(name = "aggregate_type", nullable = false)
    private String aggregateType;

    @Column(name = "event_type", nullable = false)
    private String eventType;

    @Column(name = "event_data", columnDefinition = "jsonb", nullable = false)
    private String eventData;

    @Column(name = "kafka_topic", nullable = false)
    private String kafkaTopic;

    @Column(name = "kafka_key")
    private String kafkaKey;

    @Column(name = "created_at", nullable = false)
    private Instant createdAt;

    @Column(name = "processed_at")
    private Instant processedAt;

    @Column(name = "processing_attempts", nullable = false)
    private Integer processingAttempts = 0;

    @Column(name = "last_error")
    private String lastError;

    @Column(name = "version", nullable = false)
    @Version
    private Long version = 0L;

    @Enumerated(EnumType.STRING)
    @Column(name = "status", nullable = false)
    private OutboxEventStatus status = OutboxEventStatus.PENDING;

    public enum OutboxEventStatus {
        PENDING,
        PROCESSING,
        PROCESSED,
        FAILED
    }

    @PrePersist
    public void prePersist() {
        if (createdAt == null) {
            createdAt = Instant.now();
        }
    }
}

/**
 * Business entities using outbox pattern
 */
```

```
@Entity
@Table(name = "orders")
@lombok.Data
@lombok.Builder
@lombok.NoArgsConstructor
@lombok.AllArgsConstructor
public class Order {

    @Id
    @Column(name = "order_id")
    private String orderId;

    @Column(name = "customer_id", nullable = false)
    private String customerId;

    @Column(name = "product_id", nullable = false)
    private String productId;

    @Column(name = "quantity", nullable = false)
    private Integer quantity;

    @Column(name = "total_amount", nullable = false)
    private BigDecimal totalAmount;

    @Enumerated(EnumType.STRING)
    @Column(name = "status", nullable = false)
    private OrderStatus status;

    @Column(name = "created_at", nullable = false)
    private Instant createdAt;

    @Column(name = "updated_at", nullable = false)
    private Instant updatedAt;

    @Version
    private Long version;

    public enum OrderStatus {
        PENDING,
        CONFIRMED,
        PAID,
        SHIPPED,
        DELIVERED,
        CANCELLED
    }

    @PrePersist
    @PreUpdate
    public void updateTimestamps() {
        Instant now = Instant.now();
        if (createdAt == null) {
            createdAt = now;
        }
        updatedAt = now;
    }
}

/**
 * Repository interfaces for outbox pattern
 */
@Repository
public interface OutboxEventRepository extends JpaRepository<OutboxEvent, UUID> {

    @Query("SELECT e FROM OutboxEvent e WHERE e.status = :status ORDER BY
e.createdAt ASC")
    List<OutboxEvent> findByStatusOrderByCreatedAt(@Param("status")
OutboxEvent.OutboxEventStatus status, Pageable pageable);

    @Query("SELECT e FROM OutboxEvent e WHERE e.status = 'PENDING' AND e.createdAt
<= :maxAge ORDER BY e.createdAt ASC")
    List<OutboxEvent> findPendingEventsOlderThan(@Param("maxAge") Instant maxAge,
Pageable pageable);

    @Query("SELECT e FROM OutboxEvent e WHERE e.status = 'FAILED' AND
e.processingAttempts < :maxAttempts ORDER BY e.createdAt ASC")
    List<OutboxEvent> findFailedEventsForRetry(@Param("maxAttempts") int
maxAttempts, Pageable pageable);

    @Modifying
    @Query("DELETE FROM OutboxEvent e WHERE e.status = 'PROCESSED' AND
e.processedAt <= :cutoffTime")
    int deleteProcessedEventsOlderThan(@Param("cutoffTime") Instant cutoffTime);

    @Query("SELECT COUNT(e) FROM OutboxEvent e WHERE e.status = :status")
    long countByStatus(@Param("status") OutboxEvent.OutboxEventStatus status);
}

@Repository
public interface OrderRepository extends JpaRepository<Order, String> {

    List<Order> findByCustomerIdAndStatus(String customerId, Order.OrderStatus
status);

    @Query("SELECT o FROM Order o WHERE o.updatedAt >= :since ORDER BY o.updatedAt
ASC")
    List<Order> findOrdersUpdatedSince(@Param("since") Instant since);
}

/**
 * Outbox pattern service with transactional business operations
 */
@Service
@Transactional
@lombok.extern.slf4j.Slf4j
public class OutboxPatternOrderService {

    @Autowired
    private OrderRepository orderRepository;
    @Autowired
    private OutboxEventRepository outboxEventRepository;

    @Autowired
    private ObjectMapper objectMapper;

    /**
     * Create order with outbox pattern - atomic database operation
     */
    public Order createOrder(CreateOrderRequest request) {

        String orderId = UUID.randomUUID().toString();
        log.info("Creating order with outbox pattern: orderId={}", orderId);

        try {
            // Step 1: Create business entity
            Order order = Order.builder()
                .orderId(orderId)
                .customerId(request.getCustomerId())
                .productId(request.getProductId())
                .quantity(request.getQuantity())
                .totalAmount(calculateTotalAmount(request))
                .status(Order.OrderStatus.PENDING)
                .build();

            // Save order to database
            Order savedOrder = orderRepository.save(order);

            // Step 2: Create outbox event in same transaction
            OrderCreatedEvent orderCreatedEvent = OrderCreatedEvent.builder()
                .orderId(savedOrder.getOrderId())
                .customerId(savedOrder.getCustomerId())
                .productId(savedOrder.getProductId())
                .quantity(savedOrder.getQuantity())
                .totalAmount(savedOrder.getTotalAmount())
                .createdAt(savedOrder.getCreatedAt())
                .build();

            // Store event in outbox table
            createOutboxEvent(
                savedOrder.getOrderId(),
                "Order",
                "OrderCreated",
                orderCreatedEvent,
                "order-events",
                savedOrder.getOrderId()
            );

            log.info("Order created with outbox event: orderId={}", orderId);

            return savedOrder;

        } catch (Exception e) {
            log.error("Failed to create order: orderId={}", orderId, e);
            throw new OrderCreationException("Failed to create order", e);
        }
    }

    /**
     * Update order status with outbox pattern
     */
    public Order updateOrderStatus(String orderId, Order.OrderStatus newStatus,
String reason) {

        log.info("Updating order status with outbox pattern: orderId={},
newStatus={}", orderId, newStatus);

        try {
            // Step 1: Find and update business entity
            Order order = orderRepository.findById(orderId)
                .orElseThrow(() -> new OrderNotFoundException("Order not found: "
+ orderId));

            Order.OrderStatus previousStatus = order.getStatus();
            order.setStatus(newStatus);

            Order updatedOrder = orderRepository.save(order);

            // Step 2: Create outbox event for status change
            OrderStatusChangedEvent statusChangedEvent =
OrderStatusChangedEvent.builder()
                .orderId(orderId)
                .previousStatus(previousStatus.name())
                .newStatus(newStatus.name())
                .reason(reason)
                .changedAt(updatedOrder.getUpdatedAt())
                .build();

            createOutboxEvent(
                orderId,
                "Order",
                "OrderStatusChanged",
                statusChangedEvent,
                "order-status-events",
                orderId
            );

            // Step 3: Create additional events based on status
            if (newStatus == Order.OrderStatus.PAID) {
                // Create payment confirmation event
                PaymentConfirmedEvent paymentEvent =
PaymentConfirmedEvent.builder()
                    .orderId(orderId)
                    .amount(updatedOrder.getTotalAmount())
                    .confirmedAt(updatedOrder.getUpdatedAt())
                    .build();
                createOutboxEvent(
                    orderId,
                    "Payment",
                    "PaymentConfirmed",
                    paymentEvent,
                    "payment-events",
                    orderId
                );

                // Trigger shipping process
                ShippingRequestedEvent shippingEvent = 
ShippingRequestedEvent.builder()
                    .orderId(orderId)
                    .customerId(updatedOrder.getCustomerId())
                    .productId(updatedOrder.getProductId())
                    .quantity(updatedOrder.getQuantity())
                    .requestedAt(updatedOrder.getUpdatedAt())
                    .build();

                createOutboxEvent(
                    orderId,
                    "Shipping",
                    "ShippingRequested",
                    shippingEvent,
                    "shipping-events",
                    orderId
                );
            }

            log.info("Order status updated with outbox events: orderId={}, status=
{}", orderId, newStatus);

            return updatedOrder;

        } catch (Exception e) {
            log.error("Failed to update order status: orderId={}, status={}", 
orderId, newStatus, e);
            throw new OrderUpdateException("Failed to update order status", e);
        }
    }

    /**
     * Cancel order with compensation events
     */
    public Order cancelOrder(String orderId, String cancellationReason) {

        log.info("Canceling order with outbox pattern: orderId={}, reason={}", 
orderId, cancellationReason);

        try {
            // Step 1: Update order status
            Order cancelledOrder = updateOrderStatus(orderId, 
Order.OrderStatus.CANCELLED, cancellationReason);
```

```
            // Step 2: Create compensation events
            if (cancelledOrder.getStatus() == Order.OrderStatus.PAID) {
                // Create refund event
                RefundRequestedEvent refundEvent = RefundRequestedEvent.builder()
                    .orderId(orderId)
                    .amount(cancelledOrder.getTotalAmount())
                    .reason(cancellationReason)
                    .requestedAt(Instant.now())
                    .build();

                createOutboxEvent(
                    orderId,
                    "Refund",
                    "RefundRequested",
                    refundEvent,
                    "refund-events",
                    orderId
                );
            }

            // Step 3: Create inventory restoration event
            InventoryRestorationEvent inventoryEvent =
InventoryRestorationEvent.builder()
                .orderId(orderId)
                .productId(cancelledOrder.getProductId())
                .quantity(cancelledOrder.getQuantity())
                .reason(cancellationReason)
                .requestedAt(Instant.now())
                .build();

            createOutboxEvent(
                orderId,
                "Inventory",
                "InventoryRestoration",
                inventoryEvent,
                "inventory-events",
                cancelledOrder.getProductId()
            );

            log.info("Order cancelled with compensation events: orderId={}",
orderId);

            return cancelledOrder;

        } catch (Exception e) {
            log.error("Failed to cancel order: orderId={}", orderId, e);
            throw new OrderCancellationException("Failed to cancel order", e);
        }
    }

    /**
     * Bulk order processing with outbox pattern
     */
    public List<Order> processBulkOrders(List<CreateOrderRequest> orderRequests) {
        log.info("Processing bulk orders with outbox pattern: count={}", 
orderRequests.size());

        try {
            List<Order> processedOrders = new ArrayList<>();

            // Process all orders in single transaction
            for (CreateOrderRequest request : orderRequests) {
                Order order = createOrderInternal(request);
                processedOrders.add(order);
            }

            // Create bulk processing completion event
            BulkOrderProcessingCompletedEvent bulkEvent = 
BulkOrderProcessingCompletedEvent.builder()
                .batchId(UUID.randomUUID().toString())
                .orderCount(processedOrders.size())

.orderIds(processedOrders.stream().map(Order::getOrderId).collect(Collectors.toLis
t()))
                .completedAt(Instant.now())
                .build();

            createOutboxEvent(
                bulkEvent.getBatchId(),
                "BulkProcessing",
                "BulkOrderProcessingCompleted",
                bulkEvent,
                "bulk-processing-events",
                bulkEvent.getBatchId()
            );

            log.info("Bulk order processing completed: count={}, batchId={}",
                processedOrders.size(), bulkEvent.getBatchId());

            return processedOrders;

        } catch (Exception e) {
            log.error("Failed to process bulk orders: count={}", 
orderRequests.size(), e);
            throw new BulkOrderProcessingException("Failed to process bulk 
orders", e);
        }
    }

    // Helper methods
    private void createOutboxEvent(String aggregateId, String aggregateType, 
String eventType,
                                 Object eventData, String kafkaTopic, String 
kafkaKey) {
        try {
            OutboxEvent outboxEvent = OutboxEvent.builder()
                .aggregateId(aggregateId)
                .aggregateType(aggregateType)
                .eventType(eventType)
                .eventData(objectMapper.writeValueAsString(eventData))
                .kafkaTopic(kafkaTopic)
                .kafkaKey(kafkaKey)
                .status(OutboxEvent.OutboxEventStatus.PENDING)
                .build();

            outboxEventRepository.save(outboxEvent);

            log.debug("Created outbox event: aggregateId={}, eventType={}, topic=
{}",
                aggregateId, eventType, kafkaTopic);

        } catch (Exception e) {
            log.error("Failed to create outbox event: aggregateId={}, eventType=
{}",
                aggregateId, eventType, e);
            throw new OutboxEventCreationException("Failed to create outbox
event", e);
        }
    }

    private Order createOrderInternal(CreateOrderRequest request) {
        // Internal order creation without separate transaction
        String orderId = UUID.randomUUID().toString();

        Order order = Order.builder()
            .orderId(orderId)
            .customerId(request.getCustomerId())
            .productId(request.getProductId())
            .quantity(request.getQuantity())
            .totalAmount(calculateTotalAmount(request))
            .status(Order.OrderStatus.PENDING)
            .build();

        Order savedOrder = orderRepository.save(order);

        OrderCreatedEvent orderCreatedEvent = OrderCreatedEvent.builder()
            .orderId(savedOrder.getOrderId())
            .customerId(savedOrder.getCustomerId())
            .productId(savedOrder.getProductId())
            .quantity(savedOrder.getQuantity())
            .totalAmount(savedOrder.getTotalAmount())
            .createdAt(savedOrder.getCreatedAt())
            .build();

        createOutboxEvent(
            savedOrder.getOrderId(),
            "Order",
            "OrderCreated",
            orderCreatedEvent,
            "order-events",
            savedOrder.getOrderId()
        );

        return savedOrder;
    }

    private BigDecimal calculateTotalAmount(CreateOrderRequest request) {
        // Simplified calculation - in real implementation would fetch product
prices
        return
request.getUnitPrice().multiply(BigDecimal.valueOf(request.getQuantity()));
    }
}
```

This completes Part 2 of the Spring Kafka Transactions guide, covering consumer offsets within transactions and the comprehensive outbox pattern implementation. The guide continues with comparisons, best practices, and production patterns in the final part.