Spring Kafka Transactions & Exactly Once Semantics: Complete Developer Guide

A comprehensive guide covering all aspects of Spring Kafka transactions, exactly-once semantics, producer transactions, consumer offsets management, and the transactional outbox pattern with extensive Java examples and production patterns.

Table of Contents

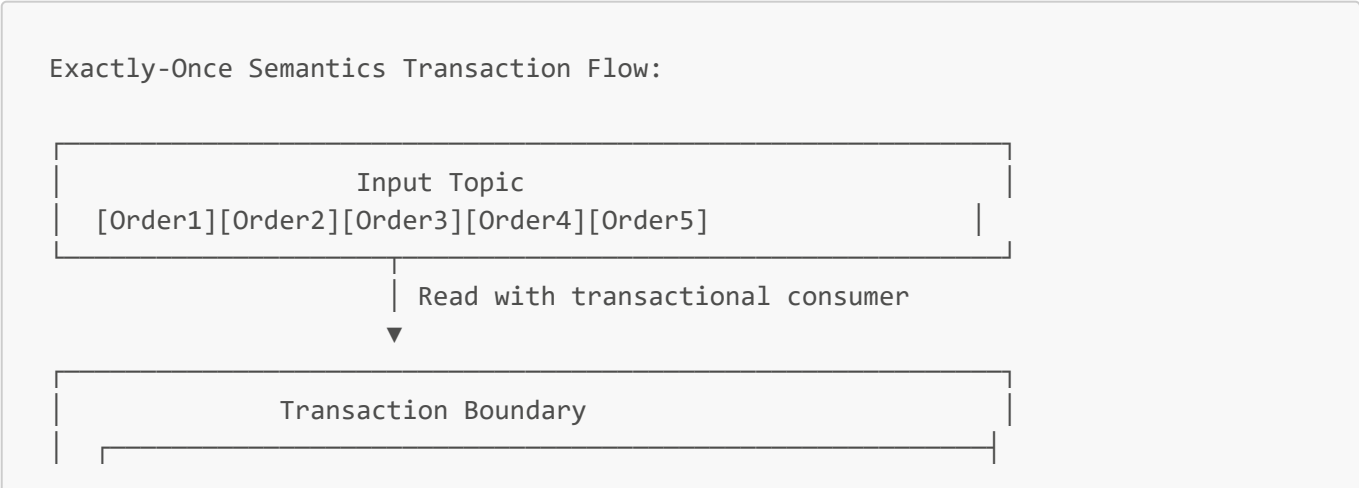
What are Kafka Transactions and Exactly-Once Semantics?

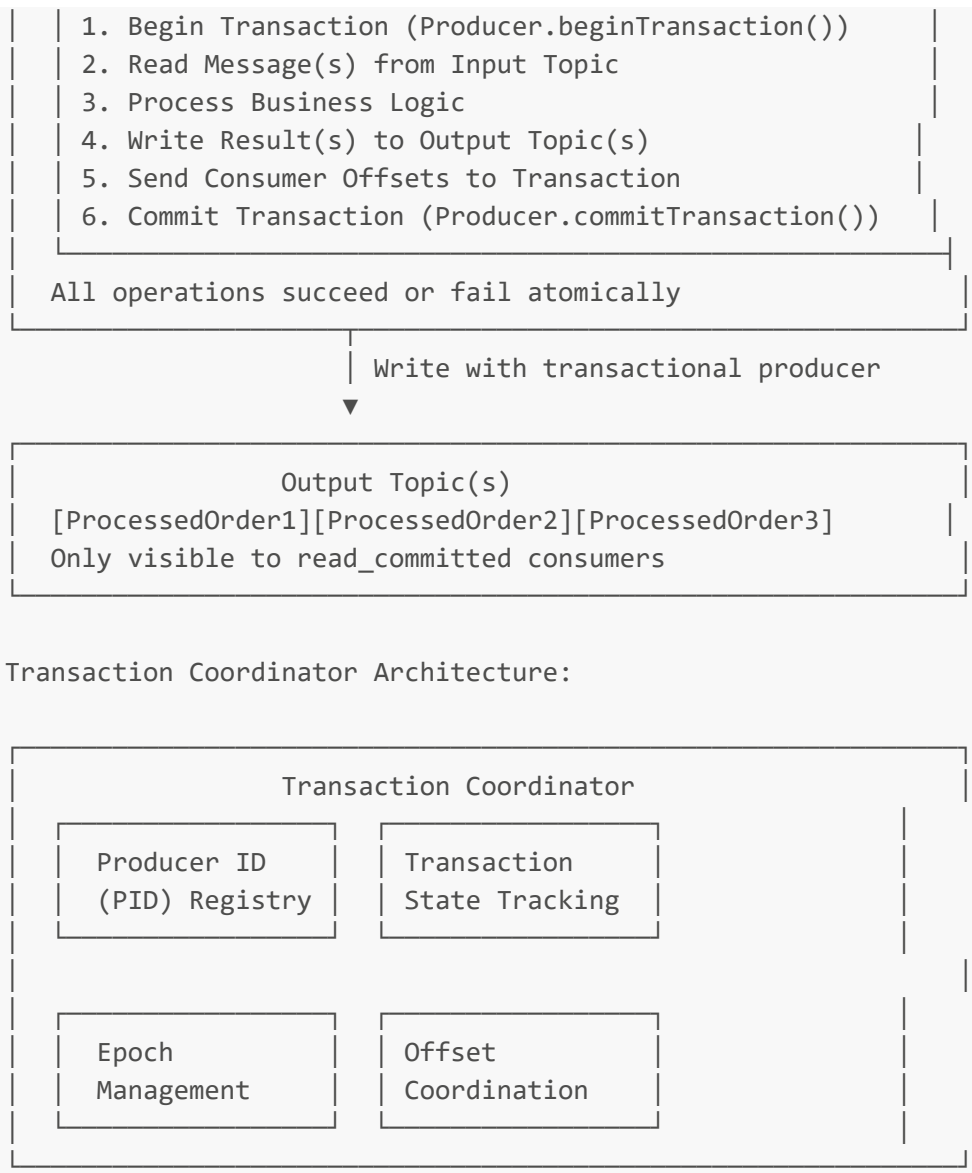
Simple Explanation: Kafka transactions enable exactly-once semantics (EOS) by ensuring that a sequence of operations (read → process → write) completes exactly once, even in the presence of failures, retries, or consumer rebalances. This prevents duplicate processing and ensures data consistency across distributed systems.

Why Exactly-Once Semantics Matter:

- **Financial Accuracy**: Prevent duplicate charges or transfers
- **Data Consistency**: Ensure accurate counts and aggregations
- **Idempotent Processing**: Same input always produces same output
- **Fault Tolerance**: Graceful handling of failures and retries
- **Complex Workflows**: Multi-step processing with rollback capability

Kafka Transaction Architecture:





Transaction Coordinator Architecture:

Transaction States:
Empty → Ongoing → PrepareCommit → CompleteCommit → CompleteAbort

---

🔄 Producer Transactions

Enabling transactional.id

Simple Explanation: The `transactional.id` is a unique identifier that enables exactly-once producer semantics. It allows Kafka to track producer state across restarts, prevent duplicate writes, and coordinate transactions with the transaction coordinator.

Why transactional.id is Critical:

- **Producer Identity**: Maintains producer identity across restarts
- **Duplicate Prevention**: Prevents duplicate messages from retries
- **Transaction Coordination**: Enables coordination with brokers
- **Exactly-Once Guarantee**: Foundation for EOS processing
- **Zombie Producer Fencing**: Prevents old producers from interfering

Basic Producer Transaction Configuration

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.kafka.core.DefaultKafkaProducerFactory;
import org.springframework.kafka.core.KafkaTemplate;
import org.springframework.kafka.core.ProducerFactory;
import org.springframework.kafka.transaction.KafkaTransactionManager;

import org.apache.kafka.clients.producer.ProducerConfig;
import org.apache.kafka.common.serialization.StringSerializer;

/**
 * Basic transactional producer configuration
 */
@Configuration
@lombok.extern.slf4j.Slf4j
public class TransactionalProducerConfiguration {

    /**
     * Transactional producer factory with transactional.id
     */
    @Bean
    public ProducerFactory<String, Object> transactionalProducerFactory() {
        Map<String, Object> props = new HashMap<>();

        // Basic Kafka configuration
        props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
StringSerializer.class);
        props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
JsonSerializer.class);

        // CRITICAL: Enable transactions with unique transactional.id
        props.put(ProducerConfig.TRANSACTIONAL_ID_CONFIG, "payment-processor-tx-
id-1");

        // Transactional settings (automatically enabled with transactional.id)
        props.put(ProducerConfig.ENABLE_IDEMPOTENCE_CONFIG, true); // Auto-enabled
        props.put(ProducerConfig.ACKS_CONFIG, "all"); // Auto-set
        props.put(ProducerConfig.RETRIES_CONFIG, Integer.MAX_VALUE); // Auto-set
        props.put(ProducerConfig.MAX_IN_FLIGHT_REQUESTS_PER_CONNECTION, 5); //
Auto-set

        // Performance optimization for transactional producers
        props.put(ProducerConfig.BATCH_SIZE_CONFIG, 65536); // 64KB
        props.put(ProducerConfig.LINGER_MS_CONFIG, 10);
        props.put(ProducerConfig.COMPRESSION_TYPE_CONFIG, "snappy");

        // Transaction timeout configuration
        props.put(ProducerConfig.TRANSACTION_TIMEOUT_CONFIG, 60000); // 60 seconds
```

```
        DefaultKafkaProducerFactory<String, Object> factory = new
DefaultKafkaProducerFactory<>(props);

        log.info("Created transactional producer factory with transactional.id:
payment-processor-tx-id-1");

        return factory;
    }

    /**
     * Transactional KafkaTemplate
     */
    @Bean
    public KafkaTemplate<String, Object> transactionalKafkaTemplate() {
        KafkaTemplate<String, Object> template = new KafkaTemplate<>
(transactionalProducerFactory());

        // Enable transactions on template
        template.setTransactionIdPrefix("payment-tx-");

        log.info("Created transactional KafkaTemplate with transaction prefix:
payment-tx-");

        return template;
    }

    /**
     * Kafka transaction manager for Spring @Transactional support
     */
    @Bean
    public KafkaTransactionManager kafkaTransactionManager() {
        return new KafkaTransactionManager(transactionalProducerFactory());
    }
}

/**
 * Comprehensive transactional producer service examples
 */
@Service
@lombok.extern.slf4j.Slf4j
public class TransactionalProducerService {

    @Autowired
    private KafkaTemplate<String, Object> transactionalKafkaTemplate;

    /**
     * Basic transactional message sending
     */
    @Transactional("kafkaTransactionManager")
    public void sendTransactionalMessage(String topic, String key, Object message)
{

        log.info("Sending transactional message: topic={}, key={}", topic, key);
```

```
        try {
            // Send message within transaction
            ListenableFuture<SendResult<String, Object>> future =
                transactionalKafkaTemplate.send(topic, key, message);

            // Add callback for monitoring
            future.addCallback(
                result -> log.info("Message sent successfully: offset={}",
                    result.getRecordMetadata().offset()),
                failure -> log.error("Failed to send message", failure)
            );

            log.info("Transactional message queued for sending");

        } catch (Exception e) {
            log.error("Error sending transactional message", e);
            throw e; // Will trigger transaction rollback
        }
    }

    /**
     * Manual transaction management with explicit control
     */
    public void sendWithManualTransaction(List<MessageData> messages) {

        log.info("Starting manual transaction for {} messages", messages.size());

        // Begin transaction explicitly
        transactionalKafkaTemplate.executeInTransaction(kafkaTemplate -> {

            try {
                // Send multiple messages in single transaction
                for (MessageData messageData : messages) {
                    kafkaTemplate.send(messageData.getTopic(),
messageData.getKey(), messageData.getValue());

                    log.debug("Queued message in transaction: topic={}, key={}",
                        messageData.getTopic(), messageData.getKey());
                }

                // Simulate business logic that might fail
                if (containsInvalidData(messages)) {
                    throw new ValidationException("Invalid data detected - rolling
back transaction");
                }

                log.info("Manual transaction completed successfully: {} messages
sent", messages.size());

                return true; // Success

            } catch (Exception e) {
                log.error("Error in manual transaction - will rollback", e);
                throw e; // Will trigger rollback
```

```
            }
        });
    }

    /**
     * Advanced transactional producer with retry logic
     */
    public void sendWithRetryLogic(String topic, String key, Object message, int
maxRetries) {

        int attempt = 0;
        Exception lastException = null;

        while (attempt < maxRetries) {
            try {
                attempt++;
                log.info("Transaction attempt {} of {} for message: key={}",
attempt, maxRetries, key);

                transactionalKafkaTemplate.executeInTransaction(kafkaTemplate -> {
                    // Send message
                    kafkaTemplate.send(topic, key, message);

                    // Simulate potential failure point
                    if (shouldSimulateFailure()) {
                        throw new TransientException("Simulated transient
failure");
                    }

                    return true;
                });

                log.info("Transaction succeeded on attempt {}: key={}", attempt,
key);
                return; // Success

            } catch (TransientException e) {
                lastException = e;
                log.warn("Transient failure on attempt {} for key={}: {}",
attempt, key, e.getMessage());

                if (attempt < maxRetries) {
                    try {
                        // Exponential backoff
                        Thread.sleep(1000 * (long) Math.pow(2, attempt - 1));
                    } catch (InterruptedException ie) {
                        Thread.currentThread().interrupt();
                        break;
                    }
                }

            } catch (Exception e) {
                log.error("Non-retryable error in transaction: key={}", key, e);
                throw e; // Don't retry for non-transient errors
```

```
            }
        }

        log.error("All transaction attempts failed for key={}: {}", key,
lastException.getMessage());
        throw new RuntimeException("Transaction failed after " + maxRetries + "
attempts", lastException);
    }

    /**
     * Batch transactional processing for high throughput
     */
    public void sendBatchTransactional(List<BatchMessage> batchMessages, int
batchSize) {

        log.info("Processing {} messages in batches of {}", batchMessages.size(),
batchSize);

        for (int i = 0; i < batchMessages.size(); i += batchSize) {
            int endIndex = Math.min(i + batchSize, batchMessages.size());
            List<BatchMessage> batch = batchMessages.subList(i, endIndex);

            log.info("Processing batch {}: {} messages", (i / batchSize) + 1,
batch.size());

            try {
                processBatchInTransaction(batch);

            } catch (Exception e) {
                log.error("Failed to process batch starting at index {}", i, e);
                throw e;
            }
        }
    }

    private void processBatchInTransaction(List<BatchMessage> batch) {
        transactionalKafkaTemplate.executeInTransaction(kafkaTemplate -> {

            try {
                // Send all messages in batch within single transaction
                for (BatchMessage message : batch) {
                    kafkaTemplate.send(message.getTopic(), message.getKey(),
message.getData());
                }

                log.debug("Batch transaction completed: {} messages",
batch.size());

                return true;

            } catch (Exception e) {
                log.error("Error in batch transaction", e);
                throw e;
            }
```

```
        });
    }

    // Helper methods
    private boolean containsInvalidData(List<MessageData> messages) {
        return messages.stream().anyMatch(msg ->
            msg.getValue() == null || msg.getTopic() == null);
    }

    private boolean shouldSimulateFailure() {
        return Math.random() < 0.1; // 10% failure rate for testing
    }
}

/**
 * Advanced transactional producer with multiple transaction managers
 */
@Configuration
@lombok.extern.slf4j.Slf4j
public class MultiTransactionManagerConfiguration {

    /**
     * Order processing transactional producer
     */
    @Bean("orderTransactionalProducerFactory")
    public ProducerFactory<String, Object> orderTransactionalProducerFactory() {
        Map<String, Object> props = new HashMap<>();

        props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
StringSerializer.class);
        props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
JsonSerializer.class);

        // Unique transactional.id for order processing
        props.put(ProducerConfig.TRANSACTIONAL_ID_CONFIG, "order-processor-tx-" +
UUID.randomUUID().toString());
        props.put(ProducerConfig.TRANSACTION_TIMEOUT_CONFIG, 30000); // 30 seconds
for orders

        return new DefaultKafkaProducerFactory<>(props);
    }

    /**
     * Payment processing transactional producer
     */
    @Bean("paymentTransactionalProducerFactory")
    public ProducerFactory<String, Object> paymentTransactionalProducerFactory() {
        Map<String, Object> props = new HashMap<>();

        props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
StringSerializer.class);
        props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
```

```
JsonSerializer.class);

        // Unique transactional.id for payment processing
        props.put(ProducerConfig.TRANSACTIONAL_ID_CONFIG, "payment-processor-tx-"
+ UUID.randomUUID().toString());
        props.put(ProducerConfig.TRANSACTION_TIMEOUT_CONFIG, 60000); // 60 seconds
for payments

        return new DefaultKafkaProducerFactory<>(props);
    }

    @Bean("orderKafkaTemplate")
    public KafkaTemplate<String, Object> orderKafkaTemplate() {
        return new KafkaTemplate<>(orderTransactionalProducerFactory());
    }

    @Bean("paymentKafkaTemplate")
    public KafkaTemplate<String, Object> paymentKafkaTemplate() {
        return new KafkaTemplate<>(paymentTransactionalProducerFactory());
    }

    @Bean("orderTransactionManager")
    public KafkaTransactionManager orderTransactionManager() {
        return new KafkaTransactionManager(orderTransactionalProducerFactory());
    }

    @Bean("paymentTransactionManager")
    public KafkaTransactionManager paymentTransactionManager() {
        return new KafkaTransactionManager(paymentTransactionalProducerFactory());
    }
}

/**
 * Service using multiple transaction managers for different business domains
 */
@Service
@lombok.extern.slf4j.Slf4j
public class MultiTransactionService {

    @Autowired
    @Qualifier("orderKafkaTemplate")
    private KafkaTemplate<String, Object> orderKafkaTemplate;

    @Autowired
    @Qualifier("paymentKafkaTemplate")
    private KafkaTemplate<String, Object> paymentKafkaTemplate;

    /**
     * Order processing with dedicated transaction manager
     */
    @Transactional("orderTransactionManager")
    public void processOrder(OrderEvent orderEvent) {

        log.info("Processing order transactionally: orderId={}",
```

```
orderEvent.getOrderId());

        try {
            // Validate order
            validateOrder(orderEvent);

            // Send order confirmation
            orderKafkaTemplate.send("order-confirmations",
orderEvent.getOrderId(), orderEvent);

            // Send inventory update
            InventoryUpdate inventoryUpdate = createInventoryUpdate(orderEvent);
            orderKafkaTemplate.send("inventory-updates", orderEvent.getOrderId(),
inventoryUpdate);

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

        } catch (Exception e) {
            log.error("Error processing order: orderId={}",
orderEvent.getOrderId(), e);
            throw e; // Trigger rollback
        }
    }

    /**
     * Payment processing with dedicated transaction manager
     */
    @Transactional("paymentTransactionManager")
    public void processPayment(PaymentEvent paymentEvent) {

        log.info("Processing payment transactionally: paymentId={}",
paymentEvent.getPaymentId());

        try {
            // Validate payment
            validatePayment(paymentEvent);

            // Send payment confirmation
            paymentKafkaTemplate.send("payment-confirmations",
paymentEvent.getPaymentId(), paymentEvent);

            // Send accounting update
            AccountingUpdate accountingUpdate =
createAccountingUpdate(paymentEvent);
            paymentKafkaTemplate.send("accounting-updates",
paymentEvent.getPaymentId(), accountingUpdate);

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

        } catch (Exception e) {
            log.error("Error processing payment: paymentId={}",
paymentEvent.getPaymentId(), e);
```

```
                throw e; // Trigger rollback
            }
        }

        // Helper methods
        private void validateOrder(OrderEvent orderEvent) {
            if (orderEvent.getOrderId() == null) {
                throw new ValidationException("Order ID cannot be null");
            }
            // Additional validation logic
        }

        private void validatePayment(PaymentEvent paymentEvent) {
            if (paymentEvent.getAmount().compareTo(BigDecimal.ZERO) <= 0) {
                throw new ValidationException("Payment amount must be positive");
            }
            // Additional validation logic
        }

        private InventoryUpdate createInventoryUpdate(OrderEvent orderEvent) {
            return InventoryUpdate.builder()
                .orderId(orderEvent.getOrderId())
                .productId(orderEvent.getProductId())
                .quantity(-orderEvent.getQuantity()) // Decrease inventory
                .timestamp(Instant.now())
                .build();
        }

        private AccountingUpdate createAccountingUpdate(PaymentEvent paymentEvent) {
            return AccountingUpdate.builder()
                .paymentId(paymentEvent.getPaymentId())
                .amount(paymentEvent.getAmount())
                .accountId(paymentEvent.getAccountId())
                .timestamp(Instant.now())
                .build();
        }
    }

    // Supporting data structures
    @lombok.Data
    @lombok.Builder
    @lombok.NoArgsConstructor
    @lombok.AllArgsConstructor
    class MessageData {
        private String topic;
        private String key;
        private Object value;
    }

    @lombok.Data
    @lombok.Builder
    @lombok.NoArgsConstructor
    @lombok.AllArgsConstructor
    class BatchMessage {
```

```
    private String topic;
    private String key;
    private Object data;
}

@lombok.Data
@lombok.Builder
@lombok.NoArgsConstructor
@lombok.AllArgsConstructor
class OrderEvent {
    private String orderId;
    private String productId;
    private Integer quantity;
    private BigDecimal amount;
    private String customerId;
    private Instant timestamp;
}

@lombok.Data
@lombok.Builder
@lombok.NoArgsConstructor
@lombok.AllArgsConstructor
class PaymentEvent {
    private String paymentId;
    private String orderId;
    private BigDecimal amount;
    private String accountId;
    private String paymentMethod;
    private Instant timestamp;
}

@lombok.Data
@lombok.Builder
@lombok.NoArgsConstructor
@lombok.AllArgsConstructor
class InventoryUpdate {
    private String orderId;
    private String productId;
    private Integer quantity;
    private Instant timestamp;
}

@lombok.Data
@lombok.Builder
@lombok.NoArgsConstructor
@lombok.AllArgsConstructor
class AccountingUpdate {
    private String paymentId;
    private BigDecimal amount;
    private String accountId;
    private Instant timestamp;
}

// Custom exceptions
```

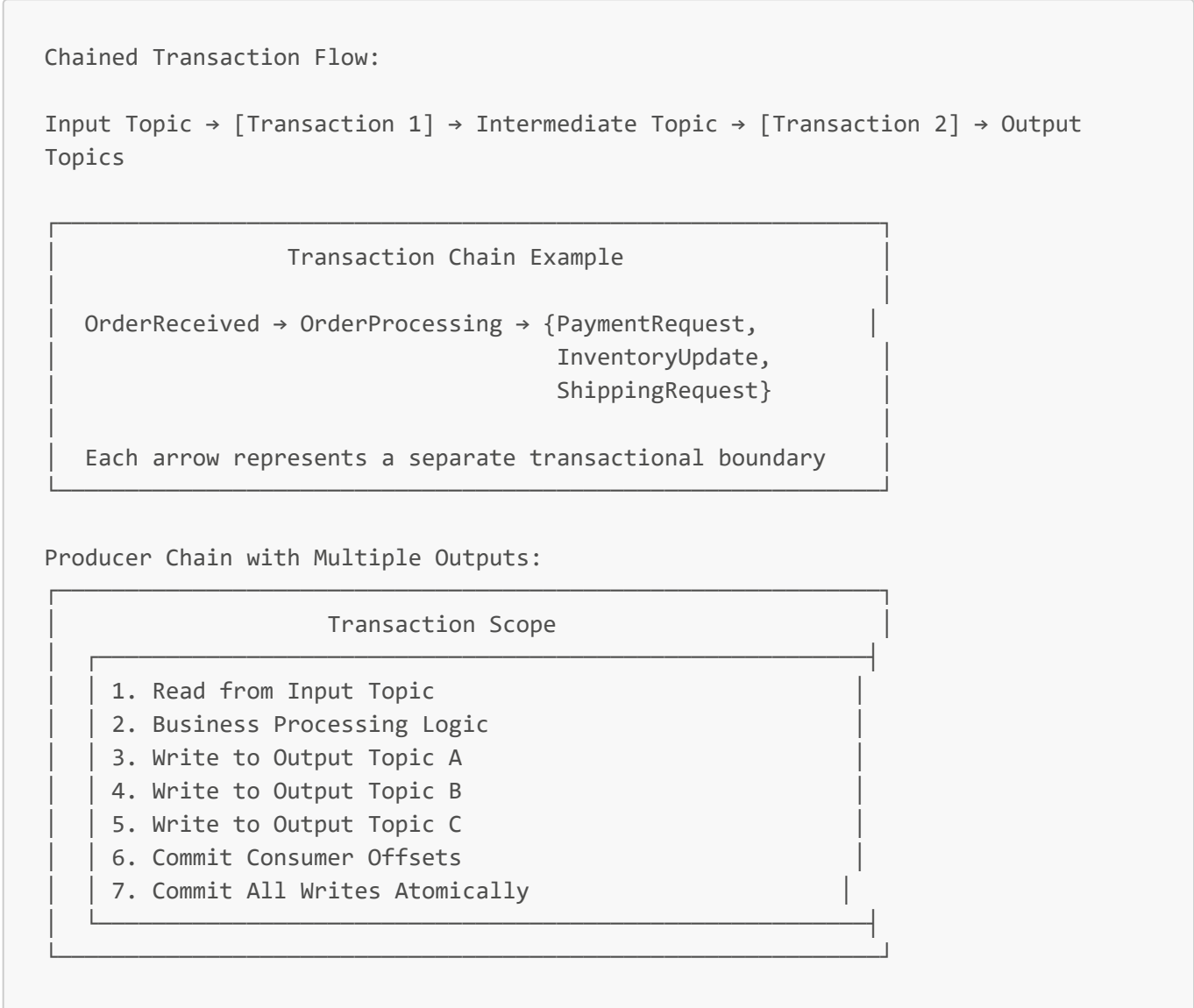
```
class ValidationException extends RuntimeException {
    public ValidationException(String message) { super(message); }
}

class TransientException extends RuntimeException {
    public TransientException(String message) { super(message); }
}
```

Chained Kafka Transactions

Simple Explanation: Chained transactions occur when one transactional operation triggers another, creating a chain of atomic operations across multiple topics and partitions. This is common in microservices architectures where processing one message results in publishing multiple related messages.

Chained Transaction Architecture:



Advanced Chained Transaction Implementation

```
/**
 * Chained transaction service for complex business flows
```

```
 */
@Service
@lombok.extern.slf4j.Slf4j
public class ChainedTransactionService {

    @Autowired
    private KafkaTemplate<String, Object> transactionalKafkaTemplate;

    @Autowired
    private OrderService orderService;

    @Autowired
    private PaymentService paymentService;

    @Autowired
    private InventoryService inventoryService;

    /**
     * Order processing chain - single transaction, multiple outputs
     */
    @Transactional("kafkaTransactionManager")
    public void processOrderChain(OrderRequest orderRequest) {

        String orderId = orderRequest.getOrderId();
        log.info("Starting order processing chain: orderId={}", orderId);

        try {
            // Step 1: Validate and create order
            OrderEvent orderEvent = orderService.createOrder(orderRequest);

            // Step 2: Send to multiple downstream topics atomically
            sendOrderConfirmation(orderEvent);
            sendPaymentRequest(orderEvent);
            sendInventoryReservation(orderEvent);
            sendShippingRequest(orderEvent);

            // Step 3: Send audit trail
            sendAuditEvent(orderEvent, "ORDER_PROCESSED");

            log.info("Order processing chain completed: orderId={}", orderId);

        } catch (Exception e) {
            log.error("Order processing chain failed: orderId={}", orderId, e);

            // Send failure audit
            sendAuditEvent(OrderEvent.builder().orderId(orderId).build(),
"ORDER_FAILED");

            throw e; // Trigger transaction rollback
        }
    }

    /**
     * Payment processing chain with conditional logic
```

```
     */
    @Transactional("kafkaTransactionManager")
    public void processPaymentChain(PaymentRequest paymentRequest) {

        String paymentId = paymentRequest.getPaymentId();
        log.info("Starting payment processing chain: paymentId={}", paymentId);

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

            // Step 2: Send different events based on payment result
            if (paymentResult.isSuccessful()) {
                sendPaymentConfirmation(paymentResult);
                sendOrderStatusUpdate(paymentResult.getOrderId(), "PAID");
                sendAccountingEntry(paymentResult);

                // If payment is for a large amount, send additional notifications
                if (paymentResult.getAmount().compareTo(new BigDecimal("1000")) >
0) {
                    sendHighValuePaymentAlert(paymentResult);
                }

            } else {
                sendPaymentFailure(paymentResult);
                sendOrderStatusUpdate(paymentResult.getOrderId(),
"PAYMENT_FAILED");

                // Retry payment if it's a retryable failure
                if (paymentResult.isRetryable()) {
                    schedulePaymentRetry(paymentRequest);
                }
            }

            // Step 3: Always send audit regardless of result
            sendAuditEvent(paymentResult, paymentResult.isSuccessful() ?
"PAYMENT_SUCCESS" : "PAYMENT_FAILED");

            log.info("Payment processing chain completed: paymentId={}, success=
{}",
                paymentId, paymentResult.isSuccessful());

        } catch (Exception e) {
            log.error("Payment processing chain failed: paymentId={}", paymentId,
e);
            throw e;
        }
    }

    /**
     * Inventory management chain with stock validation
     */
    @Transactional("kafkaTransactionManager")
```

```
    public void processInventoryChain(InventoryRequest inventoryRequest) {

        String requestId = inventoryRequest.getRequestId();
        log.info("Starting inventory processing chain: requestId={}", requestId);

        try {
            // Step 1: Check current inventory levels
            InventoryStatus currentStatus =
inventoryService.checkInventory(inventoryRequest);

            // Step 2: Process based on availability
            if (currentStatus.isAvailable()) {
                // Reserve inventory
                InventoryReservation reservation =
inventoryService.reserveInventory(inventoryRequest);

                // Send reservation confirmation
                sendInventoryReservationConfirmation(reservation);

                // Update order status
                sendOrderStatusUpdate(inventoryRequest.getOrderId(),
"INVENTORY_RESERVED");

                // Check if low stock threshold reached
                if (currentStatus.getRemainingQuantity() <
inventoryRequest.getLowStockThreshold()) {
                    sendLowStockAlert(inventoryRequest.getProductId(),
currentStatus.getRemainingQuantity());
                }

            } else {
                // Send out of stock notification
                sendOutOfStockNotification(inventoryRequest);

                // Update order status
                sendOrderStatusUpdate(inventoryRequest.getOrderId(),
"OUT_OF_STOCK");

                // Trigger restocking process
                sendRestockingRequest(inventoryRequest.getProductId());
            }

            // Step 3: Send inventory audit
            sendAuditEvent(currentStatus, "INVENTORY_PROCESSED");

            log.info("Inventory processing chain completed: requestId={}",
requestId);

        } catch (Exception e) {
            log.error("Inventory processing chain failed: requestId={}",
requestId, e);
            throw e;
        }
    }
```

```
    /**
     * Complex chained transaction with rollback scenarios
     */
    @Transactional("kafkaTransactionManager")
    public void processComplexOrderFlow(ComplexOrderRequest complexOrderRequest) {

        String flowId = complexOrderRequest.getFlowId();
        log.info("Starting complex order flow: flowId={}", flowId);

        try {
            List<String> processedSteps = new ArrayList<>();

            // Step 1: Validate customer
            CustomerValidationResult customerValidation =
validateCustomer(complexOrderRequest.getCustomerId());
            if (!customerValidation.isValid()) {
                throw new ValidationException("Customer validation failed: " +
customerValidation.getReason());
            }
            processedSteps.add("CUSTOMER_VALIDATED");

            // Step 2: Validate products and pricing
            ProductValidationResult productValidation =
validateProducts(complexOrderRequest.getItems());
            if (!productValidation.isValid()) {
                throw new ValidationException("Product validation failed: " +
productValidation.getReason());
            }
            processedSteps.add("PRODUCTS_VALIDATED");

            // Step 3: Reserve inventory for all items
            List<InventoryReservation> reservations =
reserveAllInventory(complexOrderRequest.getItems());
            processedSteps.add("INVENTORY_RESERVED");

            // Step 4: Calculate pricing and taxes
            PricingResult pricingResult = calculatePricing(complexOrderRequest);
            processedSteps.add("PRICING_CALCULATED");

            // Step 5: Process payment
            PaymentResult paymentResult = processPayment(complexOrderRequest,
pricingResult);
            if (!paymentResult.isSuccessful()) {
                throw new PaymentException("Payment failed: " +
paymentResult.getFailureReason());
            }
            processedSteps.add("PAYMENT_PROCESSED");

            // Step 6: Create final order
            ComplexOrder finalOrder = createComplexOrder(complexOrderRequest,
pricingResult, paymentResult);
            processedSteps.add("ORDER_CREATED");
```

```
            // Step 7: Send all success events
            sendComplexOrderConfirmation(finalOrder);
            sendPaymentConfirmation(paymentResult);
            sendInventoryConfirmations(reservations);
            sendShippingInitiation(finalOrder);
            sendCustomerNotification(finalOrder);

            // Step 8: Send completion audit with all processed steps
            sendComplexFlowAudit(flowId, processedSteps, "SUCCESS");

            log.info("Complex order flow completed successfully: flowId={}, steps=
{}", flowId, processedSteps);

        } catch (Exception e) {
            log.error("Complex order flow failed: flowId={}", flowId, e);

            // Send failure audit - transaction will rollback, but we log the
failure
            try {
                sendComplexFlowAudit(flowId, Collections.singletonList("FAILED"),
"ERROR: " + e.getMessage());
            } catch (Exception auditException) {
                log.error("Failed to send failure audit for flowId={}", flowId,
auditException);
            }

            throw e; // Trigger transaction rollback
        }
    }

    // Helper methods for chained transactions
    private void sendOrderConfirmation(OrderEvent orderEvent) {
        transactionalKafkaTemplate.send("order-confirmations",
orderEvent.getOrderId(), orderEvent);
        log.debug("Sent order confirmation: orderId={}", orderEvent.getOrderId());
    }

    private void sendPaymentRequest(OrderEvent orderEvent) {
        PaymentRequest paymentRequest = PaymentRequest.builder()
            .paymentId(UUID.randomUUID().toString())
            .orderId(orderEvent.getOrderId())
            .amount(orderEvent.getAmount())
            .customerId(orderEvent.getCustomerId())
            .build();

        transactionalKafkaTemplate.send("payment-requests",
paymentRequest.getPaymentId(), paymentRequest);
        log.debug("Sent payment request: paymentId={}",
paymentRequest.getPaymentId());
    }

    private void sendInventoryReservation(OrderEvent orderEvent) {
        InventoryRequest inventoryRequest = InventoryRequest.builder()
            .requestId(UUID.randomUUID().toString())
```

```
                .orderId(orderEvent.getOrderId())
                .productId(orderEvent.getProductId())
                .quantity(orderEvent.getQuantity())
                .build();

        transactionalKafkaTemplate.send("inventory-reservations",
inventoryRequest.getRequestId(), inventoryRequest);
        log.debug("Sent inventory reservation: requestId={}",
inventoryRequest.getRequestId());
    }

    private void sendShippingRequest(OrderEvent orderEvent) {
        ShippingRequest shippingRequest = ShippingRequest.builder()
                .shippingId(UUID.randomUUID().toString())
                .orderId(orderEvent.getOrderId())
                .customerId(orderEvent.getCustomerId())
                .priority("STANDARD")
                .build();

        transactionalKafkaTemplate.send("shipping-requests",
shippingRequest.getShippingId(), shippingRequest);
        log.debug("Sent shipping request: shippingId={}",
shippingRequest.getShippingId());
    }

    private void sendAuditEvent(Object eventData, String eventType) {
        AuditEvent auditEvent = AuditEvent.builder()
                .auditId(UUID.randomUUID().toString())
                .eventType(eventType)
                .eventData(eventData)
                .timestamp(Instant.now())
                .build();

        transactionalKafkaTemplate.send("audit-events", auditEvent.getAuditId(),
auditEvent);
        log.debug("Sent audit event: auditId={}, eventType={}",
auditEvent.getAuditId(), eventType);
    }

    private void sendOrderStatusUpdate(String orderId, String status) {
        OrderStatusUpdate statusUpdate = OrderStatusUpdate.builder()
                .orderId(orderId)
                .status(status)
                .timestamp(Instant.now())
                .build();

        transactionalKafkaTemplate.send("order-status-updates", orderId,
statusUpdate);
        log.debug("Sent order status update: orderId={}, status={}", orderId,
status);
    }

    private void sendPaymentConfirmation(PaymentResult paymentResult) {
        PaymentConfirmation confirmation = PaymentConfirmation.builder()
```

```
                .paymentId(paymentResult.getPaymentId())
                .orderId(paymentResult.getOrderId())
                .amount(paymentResult.getAmount())
                .confirmationNumber(paymentResult.getConfirmationNumber())
                .timestamp(Instant.now())
                .build();

        transactionalKafkaTemplate.send("payment-confirmations",
paymentResult.getPaymentId(), confirmation);
        log.debug("Sent payment confirmation: paymentId={}",
paymentResult.getPaymentId());
    }

    private void sendAccountingEntry(PaymentResult paymentResult) {
        AccountingEntry entry = AccountingEntry.builder()
                .entryId(UUID.randomUUID().toString())
                .paymentId(paymentResult.getPaymentId())
                .amount(paymentResult.getAmount())
                .accountId(paymentResult.getAccountId())
                .entryType("DEBIT")
                .timestamp(Instant.now())
                .build();

        transactionalKafkaTemplate.send("accounting-entries", entry.getEntryId(),
entry);
        log.debug("Sent accounting entry: entryId={}", entry.getEntryId());
    }

    private void sendHighValuePaymentAlert(PaymentResult paymentResult) {
        HighValueAlert alert = HighValueAlert.builder()
                .alertId(UUID.randomUUID().toString())
                .paymentId(paymentResult.getPaymentId())
                .amount(paymentResult.getAmount())
                .alertType("HIGH_VALUE_PAYMENT")
                .timestamp(Instant.now())
                .build();

        transactionalKafkaTemplate.send("high-value-alerts", alert.getAlertId(),
alert);
        log.debug("Sent high value payment alert: alertId={}",
alert.getAlertId());
    }

    private void sendPaymentFailure(PaymentResult paymentResult) {
        PaymentFailure failure = PaymentFailure.builder()
                .paymentId(paymentResult.getPaymentId())
                .orderId(paymentResult.getOrderId())
                .failureReason(paymentResult.getFailureReason())
                .isRetryable(paymentResult.isRetryable())
                .timestamp(Instant.now())
                .build();

        transactionalKafkaTemplate.send("payment-failures",
paymentResult.getPaymentId(), failure);
```

```
            log.debug("Sent payment failure: paymentId={}",
paymentResult.getPaymentId());
        }

        private void schedulePaymentRetry(PaymentRequest paymentRequest) {
            PaymentRetrySchedule retrySchedule = PaymentRetrySchedule.builder()
                .scheduleId(UUID.randomUUID().toString())
                .paymentId(paymentRequest.getPaymentId())
                .originalPaymentRequest(paymentRequest)
                .retryAfter(Instant.now().plus(Duration.ofMinutes(15)))
                .maxRetries(3)
                .build();

            transactionalKafkaTemplate.send("payment-retry-schedule",
retrySchedule.getScheduleId(), retrySchedule);
            log.debug("Scheduled payment retry: scheduleId={}",
retrySchedule.getScheduleId());
        }

        // Additional helper methods for complex flow
        private CustomerValidationResult validateCustomer(String customerId) {
            // Implementation would validate customer
            return CustomerValidationResult.builder()
                .customerId(customerId)
                .valid(true)
                .build();
        }

        private ProductValidationResult validateProducts(List<OrderItem> items) {
            // Implementation would validate products
            return ProductValidationResult.builder()
                .valid(true)
                .build();
        }

        private List<InventoryReservation> reserveAllInventory(List<OrderItem> items)
{
            // Implementation would reserve inventory
            return items.stream()
                .map(item -> InventoryReservation.builder()
                    .reservationId(UUID.randomUUID().toString())
                    .productId(item.getProductId())
                    .quantity(item.getQuantity())
                    .build())
                .collect(Collectors.toList());
        }

        private PricingResult calculatePricing(ComplexOrderRequest request) {
            // Implementation would calculate pricing
            BigDecimal total = request.getItems().stream()
                .map(item ->
item.getPrice().multiply(BigDecimal.valueOf(item.getQuantity())))
                .reduce(BigDecimal.ZERO, BigDecimal::add);
```

```
            return PricingResult.builder()
                .totalAmount(total)
                .taxAmount(total.multiply(BigDecimal.valueOf(0.08)))
                .finalAmount(total.multiply(BigDecimal.valueOf(1.08)))
                .build();
        }

        private PaymentResult processPayment(ComplexOrderRequest request,
    PricingResult pricing) {
            // Implementation would process payment
            return PaymentResult.builder()
                .paymentId(UUID.randomUUID().toString())
                .orderId(request.getFlowId())
                .amount(pricing.getFinalAmount())
                .successful(true)
                .confirmationNumber("CONF-" + System.currentTimeMillis())
                .build();
        }

        private ComplexOrder createComplexOrder(ComplexOrderRequest request,
    PricingResult pricing, PaymentResult payment) {
            return ComplexOrder.builder()
                .orderId(request.getFlowId())
                .customerId(request.getCustomerId())
                .items(request.getItems())
                .totalAmount(pricing.getFinalAmount())
                .paymentId(payment.getPaymentId())
                .status("CONFIRMED")
                .createdAt(Instant.now())
                .build();
        }

        private void sendComplexOrderConfirmation(ComplexOrder order) {
            transactionalKafkaTemplate.send("complex-order-confirmations",
    order.getOrderId(), order);
            log.debug("Sent complex order confirmation: orderId={}",
    order.getOrderId());
        }

        private void sendInventoryConfirmations(List<InventoryReservation>
    reservations) {
            for (InventoryReservation reservation : reservations) {
                transactionalKafkaTemplate.send("inventory-confirmations",
    reservation.getReservationId(), reservation);
            }
            log.debug("Sent {} inventory confirmations", reservations.size());
        }

        private void sendShippingInitiation(ComplexOrder order) {
            ShippingInitiation initiation = ShippingInitiation.builder()
                .shippingId(UUID.randomUUID().toString())
                .orderId(order.getOrderId())
                .customerId(order.getCustomerId())
                .items(order.getItems())
```

```
            .build();

        transactionalKafkaTemplate.send("shipping-initiation",
initiation.getShippingId(), initiation);
        log.debug("Sent shipping initiation: shippingId={}",
initiation.getShippingId());
    }

    private void sendCustomerNotification(ComplexOrder order) {
        CustomerNotification notification = CustomerNotification.builder()
            .notificationId(UUID.randomUUID().toString())
            .customerId(order.getCustomerId())
            .orderId(order.getOrderId())
            .notificationType("ORDER_CONFIRMATION")
            .message("Your order has been confirmed and will be processed
shortly")
            .build();

        transactionalKafkaTemplate.send("customer-notifications",
notification.getNotificationId(), notification);
        log.debug("Sent customer notification: notificationId={}",
notification.getNotificationId());
    }

    private void sendComplexFlowAudit(String flowId, List<String> processedSteps,
String outcome) {
        ComplexFlowAudit audit = ComplexFlowAudit.builder()
            .auditId(UUID.randomUUID().toString())
            .flowId(flowId)
            .processedSteps(processedSteps)
            .outcome(outcome)
            .timestamp(Instant.now())
            .build();

        transactionalKafkaTemplate.send("complex-flow-audits", audit.getAuditId(),
audit);
        log.debug("Sent complex flow audit: auditId={}, outcome={}",
audit.getAuditId(), outcome);
    }
}
```

This completes Part 1 of the comprehensive Spring Kafka Transactions guide, covering producer transactions, transactional.id configuration, and chained transaction patterns. The guide continues with consumer offsets and the outbox pattern in the next part.