Spring Kafka Producer Side: Complete Developer Guide

A comprehensive guide covering Spring Kafka producer implementation, from KafkaTemplate basics to advanced transactional patterns with extensive Java examples and best practices.

Table of Contents

KafkaTemplate

What is KafkaTemplate?

Simple Explanation: KafkaTemplate is Spring Kafka's central class for sending messages to Kafka topics, providing a high-level abstraction over the native KafkaProducer with Spring-style configuration, error handling, and integration capabilities.

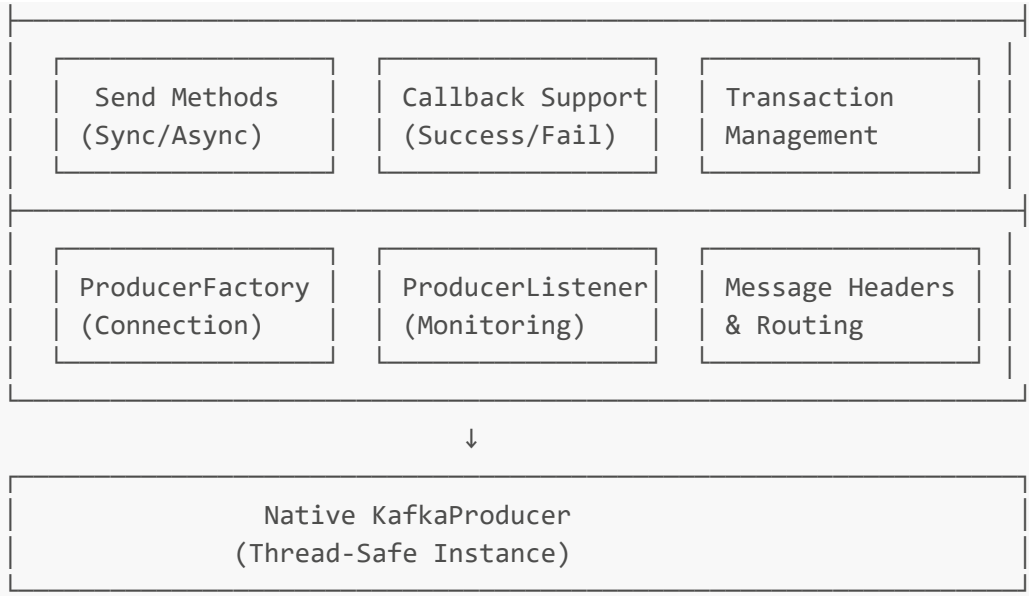
Problem It Solves:

- **Boilerplate Reduction**: Eliminates repetitive producer setup and lifecycle management
- **Spring Integration**: Seamless integration with Spring's dependency injection and transaction management
- **Error Handling**: Built-in error handling with callback mechanisms
- **Resource Management**: Automatic connection pooling and thread safety
- **Serialization**: Simplified message serialization and deserialization

Internal Architecture:

```
KafkaTemplate Architecture:

┌─────────────────────────────────────────────────────────────────┐
│                    KafkaTemplate                                │
```



Synchronous vs Asynchronous Send

Comprehensive Send Pattern Implementations

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.kafka.core.KafkaTemplate;
import org.springframework.kafka.support.SendResult;
import org.springframework.kafka.support.KafkaHeaders;
import org.springframework.messaging.Message;
import org.springframework.messaging.support.MessageBuilder;
import org.springframework.stereotype.Service;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import java.util.concurrent.CompletableFuture;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.TimeoutException;
import java.time.Instant;
import java.util.UUID;

/**
 * Comprehensive demonstration of synchronous vs asynchronous message sending
patterns
 */
@Service
public class KafkaProducerService {

    private static final Logger logger =
LoggerFactory.getLogger(KafkaProducerService.class);

    @Autowired
    private KafkaTemplate<String, Object> kafkaTemplate;
```

```
    private static final String ORDERS_TOPIC = "orders";
    private static final String NOTIFICATIONS_TOPIC = "notifications";
    private static final String EVENTS_TOPIC = "events";

    // ========================================
    // SYNCHRONOUS SEND PATTERNS
    // ========================================

    /**
     * Basic synchronous send - blocks until completion
     * Use for: Critical operations requiring immediate confirmation
     */
    public void sendSynchronously(String topic, String key, Object message) {
        try {
            logger.info("Sending message synchronously to topic: {}", topic);
            long startTime = System.currentTimeMillis();

            // Blocks until message is sent and acknowledged
            SendResult<String, Object> result = kafkaTemplate.send(topic, key,
message).get();

            long duration = System.currentTimeMillis() - startTime;

            logger.info("Message sent successfully in {}ms: topic={}, partition=
{}, offset={}",
                duration,
                result.getRecordMetadata().topic(),
                result.getRecordMetadata().partition(),
                result.getRecordMetadata().offset());

        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
            logger.error("Message sending interrupted for topic: {}", topic, e);
            throw new RuntimeException("Sending interrupted", e);
        } catch (ExecutionException e) {
            logger.error("Failed to send message to topic: {}", topic,
e.getCause());
            throw new RuntimeException("Sending failed", e.getCause());
        }
    }

    /**
     * Synchronous send with timeout - prevents indefinite blocking
     * Use for: Operations requiring confirmation within a time limit
     */
    public boolean sendSynchronouslyWithTimeout(String topic, String key, Object
message,
                                                long timeoutMs) {
        try {
            logger.info("Sending message synchronously with {}ms timeout to topic:
{}",
                timeoutMs, topic);

            long startTime = System.currentTimeMillis();
```

```
            // Blocks until completion or timeout
            SendResult<String, Object> result = kafkaTemplate.send(topic, key,
message)
                .get(timeoutMs, TimeUnit.MILLISECONDS);

            long duration = System.currentTimeMillis() - startTime;

            logger.info("Message sent within timeout ({}ms): topic={}, partition=
{}, offset={}",
                duration,
                result.getRecordMetadata().topic(),
                result.getRecordMetadata().partition(),
                result.getRecordMetadata().offset());

            return true;

        } catch (TimeoutException e) {
            logger.error("Message sending timed out after {}ms for topic: {}",
timeoutMs, topic);
            return false;
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
            logger.error("Message sending interrupted for topic: {}", topic, e);
            return false;
        } catch (ExecutionException e) {
            logger.error("Failed to send message to topic: {}", topic,
e.getCause());
            return false;
        }
    }

    /**
     * Batch synchronous send - send multiple messages in sequence
     * Use for: When you need confirmation of each message individually
     */
    public void sendBatchSynchronously(java.util.List<OrderEvent> orders) {
        logger.info("Sending batch of {} orders synchronously", orders.size());

        int successCount = 0;
        int failureCount = 0;
        long totalStartTime = System.currentTimeMillis();

        for (OrderEvent order : orders) {
            try {
                SendResult<String, Object> result = kafkaTemplate.send(
                    ORDERS_TOPIC, order.getOrderId(), order).get(5000,
TimeUnit.MILLISECONDS);

                successCount++;
                logger.debug("Order sent: {} to partition: {}, offset: {}",
                    order.getOrderId(),
                    result.getRecordMetadata().partition(),
                    result.getRecordMetadata().offset());
```

```
            } catch (Exception e) {
                failureCount++;
                logger.error("Failed to send order: {}", order.getOrderId(), e);

                // Could implement retry logic or dead letter handling here
                handleSendFailure(order, e);
            }
        }

        long totalDuration = System.currentTimeMillis() - totalStartTime;
        logger.info("Batch synchronous send completed in {}ms: {} successful, {}
failed",
            totalDuration, successCount, failureCount);
    }

    // =======================================
    // ASYNCHRONOUS SEND PATTERNS
    // =======================================

    /**
     * Basic asynchronous send - fire and forget
     * Use for: High-throughput scenarios where immediate confirmation isn't
critical
     */
    public void sendAsynchronously(String topic, String key, Object message) {
        logger.info("Sending message asynchronously to topic: {}", topic);

        CompletableFuture<SendResult<String, Object>> future =
            kafkaTemplate.send(topic, key, message);

        // Returns immediately - doesn't block
        logger.info("Message send initiated for topic: {} (non-blocking)", topic);
    }

    /**
     * Asynchronous send with callback - handle success/failure
     * Use for: When you need to react to send results but don't want to block
     */
    public void sendAsynchronouslyWithCallback(String topic, String key, Object
message) {
        logger.info("Sending message asynchronously with callback to topic: {}",
topic);

        long startTime = System.currentTimeMillis();

        CompletableFuture<SendResult<String, Object>> future =
            kafkaTemplate.send(topic, key, message);

        future.whenComplete((result, ex) -> {
            long duration = System.currentTimeMillis() - startTime;

            if (ex != null) {
                logger.error("Async message send failed after {}ms for topic: {}",
```

```
                    duration, topic, ex);
                handleAsyncSendFailure(topic, key, message, ex);
            } else {
                logger.info("Async message sent successfully after {}ms: topic={},
partition={}, offset={}",
                    duration,
                    result.getRecordMetadata().topic(),
                    result.getRecordMetadata().partition(),
                    result.getRecordMetadata().offset());
                handleAsyncSendSuccess(result);
            }
        });

        logger.info("Async send callback registered for topic: {} (non-blocking)",
topic);
    }

    /**
     * Advanced asynchronous pattern with timeout and fallback
     * Use for: Critical async operations that need timeout handling
     */
    public void sendAsynchronouslyWithTimeoutAndFallback(String topic, String key,
                                                         Object message, long
timeoutMs) {
        logger.info("Sending message asynchronously with timeout to topic: {}",
topic);

        CompletableFuture<SendResult<String, Object>> future =
            kafkaTemplate.send(topic, key, message);

        // Add timeout handling
        CompletableFuture<SendResult<String, Object>> timeoutFuture =
future.orTimeout(timeoutMs, TimeUnit.MILLISECONDS);

        timeoutFuture.whenComplete((result, ex) -> {
            if (ex != null) {
                if (ex instanceof TimeoutException) {
                    logger.error("Async message send timed out after {}ms for
topic: {}",
                        timeoutMs, topic);
                    handleSendTimeout(topic, key, message);
                } else {
                    logger.error("Async message send failed for topic: {}", topic,
ex);
                    handleAsyncSendFailure(topic, key, message, ex);
                }

                // Implement fallback logic
                implementFallbackStrategy(topic, key, message, ex);
            } else {
                logger.info("Async message sent successfully: topic={}, partition=
{}, offset={}",
                    result.getRecordMetadata().topic(),
                    result.getRecordMetadata().partition(),
```

```
                    result.getRecordMetadata().offset());
            }
        });
    }

    /**
     * Batch asynchronous send - send multiple messages concurrently
     * Use for: High-throughput scenarios with batch processing
     */
    public CompletableFuture<BatchSendResult>
sendBatchAsynchronously(java.util.List<OrderEvent> orders) {
        logger.info("Sending batch of {} orders asynchronously", orders.size());

        java.util.List<CompletableFuture<SendResult<String, Object>>> futures =
            new java.util.ArrayList<>();

        // Send all messages asynchronously
        for (OrderEvent order : orders) {
            CompletableFuture<SendResult<String, Object>> future =
                kafkaTemplate.send(ORDERS_TOPIC, order.getOrderId(), order);
            futures.add(future);
        }

        // Combine all futures into a single future
        CompletableFuture<Void> allFutures = CompletableFuture.allOf(
            futures.toArray(new CompletableFuture[0]));

        // Return a future that completes when all sends are done
        return allFutures.handle((result, ex) -> {
            BatchSendResult batchResult = new BatchSendResult();

            for (int i = 0; i < futures.size(); i++) {
                try {
                    SendResult<String, Object> sendResult = futures.get(i).get();
                    batchResult.addSuccess(orders.get(i), sendResult);
                } catch (Exception e) {
                    batchResult.addFailure(orders.get(i), e);
                }
            }

            logger.info("Batch async send completed: {} successful, {} failed",
                batchResult.getSuccessCount(), batchResult.getFailureCount());

            return batchResult;
        });
    }

    /**
     * Request-Reply pattern - synchronous communication over Kafka
     * Use for: When you need synchronous request-response semantics
     */
    public String sendAndReceive(String requestTopic, String request, long
timeoutMs)
            throws Exception {
```

```
        logger.info("Sending request-reply message to topic: {}", requestTopic);

        // Create unique correlation ID
        String correlationId = UUID.randomUUID().toString();

        // Build message with reply headers
        Message<String> message = MessageBuilder
            .withPayload(request)
            .setHeader(KafkaHeaders.TOPIC, requestTopic)
            .setHeader(KafkaHeaders.REPLY_TOPIC, "reply-topic")
            .setHeader(KafkaHeaders.CORRELATION_ID, correlationId)
            .setHeader("timestamp", System.currentTimeMillis())
            .build();

        try {
            // This would require ReplyingKafkaTemplate in real implementation
            CompletableFuture<SendResult<String, Object>> future =
kafkaTemplate.send(message);

            SendResult<String, Object> result = future.get(timeoutMs,
TimeUnit.MILLISECONDS);

            logger.info("Request sent successfully, correlation ID: {}",
correlationId);

            // In real implementation, would wait for reply message
            return "Reply for correlation ID: " + correlationId;

        } catch (TimeoutException e) {
            logger.error("Request-reply timed out after {}ms", timeoutMs);
            throw new Exception("Request-reply timeout", e);
        }
    }

    // ========================================
    // HELPER METHODS AND CALLBACKS
    // ========================================

    private void handleSendFailure(OrderEvent order, Exception e) {
        logger.error("Handling send failure for order: {}", order.getOrderId(),
e);

        // Could implement:
        // - Dead letter queue
        // - Retry mechanism
        // - Alert notification
        // - Database logging
    }

    private void handleAsyncSendSuccess(SendResult<String, Object> result) {
        // Update metrics, logs, monitoring
        logger.debug("Async send success handled: partition={}, offset={}",
            result.getRecordMetadata().partition(),
```

```
                result.getRecordMetadata().offset());
        }

        private void handleAsyncSendFailure(String topic, String key, Object message,
Throwable ex) {
            logger.error("Handling async send failure: topic={}, key={}", topic, key,
ex);

            // Implement failure handling:
            // - Retry with exponential backoff
            // - Send to dead letter topic
            // - Alert operations team
            // - Store in database for manual processing
        }

        private void handleSendTimeout(String topic, String key, Object message) {
            logger.warn("Send timeout occurred: topic={}, key={}", topic, key);

            // Implement timeout handling:
            // - Cancel operation
            // - Try alternative approach
            // - Store for later retry
        }

        private void implementFallbackStrategy(String topic, String key, Object
message, Throwable ex) {
            logger.info("Implementing fallback strategy for topic: {}", topic);

            // Fallback options:
            // - Send to alternative topic
            // - Store in local database
            // - Queue for later processing
            // - Use different messaging system
        }
    }

    /**
     * Result class for batch async operations
     */
    class BatchSendResult {
        private int successCount = 0;
        private int failureCount = 0;
        private java.util.List<OrderEvent> successfulOrders = new
java.util.ArrayList<>();
        private java.util.List<OrderEvent> failedOrders = new java.util.ArrayList<>();

        public void addSuccess(OrderEvent order, SendResult<String, Object> result) {
            successCount++;
            successfulOrders.add(order);
        }

        public void addFailure(OrderEvent order, Exception e) {
            failureCount++;
            failedOrders.add(order);
```

```
    }

    // Getters
    public int getSuccessCount() { return successCount; }
    public int getFailureCount() { return failureCount; }
    public java.util.List<OrderEvent> getSuccessfulOrders() { return
successfulOrders; }
    public java.util.List<OrderEvent> getFailedOrders() { return failedOrders; }
}

/**
 * Domain object for examples
 */
@lombok.Data
@lombok.Builder
@lombok.NoArgsConstructor
@lombok.AllArgsConstructor
class OrderEvent {
    private String orderId;
    private String customerId;
    private java.math.BigDecimal amount;
    private String status;
    private Instant timestamp;

    public static OrderEvent sample() {
        return OrderEvent.builder()
            .orderId("ORDER-" + UUID.randomUUID())
            .customerId("CUSTOMER-123")
            .amount(new java.math.BigDecimal("99.99"))
            .status("PENDING")
            .timestamp(Instant.now())
            .build();
    }
}
```

Callback Handling

Advanced Callback Patterns and Error Handling

```
import org.springframework.kafka.support.ProducerListener;
import org.springframework.kafka.core.KafkaProducerException;
import org.springframework.util.concurrent.ListenableFutureCallback;
import org.springframework.retry.annotation.Backoff;
import org.springframework.retry.annotation.Retryable;

import java.util.concurrent.atomic.AtomicLong;
import java.util.concurrent.ConcurrentHashMap;
import java.util.Map;

/**
 * Advanced callback handling patterns for Spring Kafka producers
```

```
 */
@Service
public class KafkaCallbackService {

    private static final Logger logger =
LoggerFactory.getLogger(KafkaCallbackService.class);

    @Autowired
    private KafkaTemplate<String, Object> kafkaTemplate;

    // Metrics tracking
    private final AtomicLong successCount = new AtomicLong(0);
    private final AtomicLong failureCount = new AtomicLong(0);
    private final Map<String, Long> topicMetrics = new ConcurrentHashMap<>();

    /**
     * ProducerListener implementation for global callback handling
     */
    @Component
    public static class CustomProducerListener implements ProducerListener<String,
Object> {

        private static final Logger logger =
LoggerFactory.getLogger(CustomProducerListener.class);

        @Override
        public void onSuccess(ProducerRecord<String, Object> producerRecord,
                              RecordMetadata recordMetadata) {

            logger.info("✅ Message sent successfully: topic={}, partition={},
offset={}, key={}",
                recordMetadata.topic(),
                recordMetadata.partition(),
                recordMetadata.offset(),
                producerRecord.key());

            // Update success metrics
            updateSuccessMetrics(producerRecord, recordMetadata);

            // Trigger success-specific logic
            handleGlobalSuccess(producerRecord, recordMetadata);
        }

        @Override
        public void onError(ProducerRecord<String, Object> producerRecord,
                            RecordMetadata recordMetadata, Exception exception) {

            logger.error("❌ Message send failed: topic={}, key={}, error={}",
                producerRecord.topic(),
                producerRecord.key(),
                exception.getMessage(), exception);

            // Update error metrics
            updateErrorMetrics(producerRecord, exception);
```

```
            // Handle different types of errors
            if (exception instanceof KafkaProducerException) {
                handleKafkaProducerException((KafkaProducerException) exception);
            } else if (exception instanceof
org.apache.kafka.common.errors.TimeoutException) {
                handleTimeoutException(producerRecord, exception);
            } else if (exception instanceof
org.apache.kafka.common.errors.RetriableException) {
                handleRetriableException(producerRecord, exception);
            } else {
                handleUnknownException(producerRecord, exception);
            }
        }

        private void updateSuccessMetrics(ProducerRecord<String, Object> record,
                                          RecordMetadata metadata) {
            // Implement metrics collection
            logger.debug("Updating success metrics for topic: {}",
record.topic());
        }

        private void updateErrorMetrics(ProducerRecord<String, Object> record,
Exception ex) {
            // Implement error metrics collection
            logger.debug("Updating error metrics for topic: {}", record.topic());
        }

        private void handleGlobalSuccess(ProducerRecord<String, Object> record,
                                         RecordMetadata metadata) {
            // Global success handling logic
            // - Update dashboards
            // - Trigger downstream processing
            // - Update audit logs
        }

        private void handleKafkaProducerException(KafkaProducerException ex) {
            logger.error("Kafka producer specific error: {}", ex.getMessage());
            // Handle producer-specific errors
        }

        private void handleTimeoutException(ProducerRecord<String, Object> record,
Exception ex) {
            logger.error("Timeout sending to topic: {}", record.topic());
            // Handle timeout-specific logic
        }

        private void handleRetriableException(ProducerRecord<String, Object>
record, Exception ex) {
            logger.warn("Retriable exception for topic: {}", record.topic());
            // Could trigger retry mechanism
        }

        private void handleUnknownException(ProducerRecord<String, Object> record,
```

```
Exception ex) {
            logger.error("Unknown exception for topic: {}", record.topic(), ex);
            // Handle unknown errors - alert, log, etc.
        }
    }

    /**
     * Custom callback implementation for specific message handling
     */
    public class OrderEventCallback implements
ListenableFutureCallback<SendResult<String, Object>> {

        private final OrderEvent originalOrder;
        private final String correlationId;
        private final long startTime;

        public OrderEventCallback(OrderEvent order, String correlationId) {
            this.originalOrder = order;
            this.correlationId = correlationId;
            this.startTime = System.currentTimeMillis();
        }

        @Override
        public void onSuccess(SendResult<String, Object> result) {
            long duration = System.currentTimeMillis() - startTime;
            successCount.incrementAndGet();

            logger.info("✅ Order event sent successfully in {}ms: orderId={},
correlationId={}, partition={}, offset={}",
                duration,
                originalOrder.getOrderId(),
                correlationId,
                result.getRecordMetadata().partition(),
                result.getRecordMetadata().offset());

            // Order-specific success handling
            handleOrderSendSuccess(originalOrder, result, duration);

            // Update metrics
            updateTopicMetrics(result.getRecordMetadata().topic(), duration);

            // Trigger downstream processing
            triggerDownstreamProcessing(originalOrder, result);
        }

        @Override
        public void onFailure(Throwable ex) {
            long duration = System.currentTimeMillis() - startTime;
            failureCount.incrementAndGet();

            logger.error("❌ Order event send failed after {}ms: orderId={},
correlationId={}, error={}",
                duration,
                originalOrder.getOrderId(),
```

```
                correlationId,
                ex.getMessage(), ex);

            // Order-specific failure handling
            handleOrderSendFailure(originalOrder, ex, duration);

            // Implement retry logic if appropriate
            if (shouldRetry(ex)) {
                retryOrderSend(originalOrder, correlationId);
            } else {
                // Send to dead letter queue
                sendToDeadLetterQueue(originalOrder, ex);
            }

            // Alert if critical
            if (isCriticalOrder(originalOrder)) {
                sendCriticalOrderAlert(originalOrder, ex);
            }
        }

        private void handleOrderSendSuccess(OrderEvent order, SendResult<String,
Object> result, long duration) {
            // Business logic for successful order send
            logger.debug("Processing successful order send: {}",
order.getOrderId());

            // Could update order status in database
            // Could trigger inventory reservation
            // Could send customer notification
        }

        private void handleOrderSendFailure(OrderEvent order, Throwable ex, long
duration) {
            // Business logic for failed order send
            logger.debug("Processing failed order send: {}", order.getOrderId());

            // Could update order status to failed
            // Could trigger compensating actions
            // Could alert customer service
        }

        private void triggerDownstreamProcessing(OrderEvent order,
SendResult<String, Object> result) {
            // Trigger next steps in the order processing pipeline
            logger.debug("Triggering downstream processing for order: {}",
order.getOrderId());
        }

        private boolean shouldRetry(Throwable ex) {
            // Determine if the error is retryable
            return ex instanceof org.apache.kafka.common.errors.RetriableException
||
                   ex instanceof TimeoutException ||
                   ex.getMessage().contains("timeout");
```

```
        }

        private void retryOrderSend(OrderEvent order, String correlationId) {
            logger.info("Retrying order send: orderId={}, correlationId={}",
                order.getOrderId(), correlationId);

            // Implement exponential backoff retry
            // Could use Spring Retry or custom implementation
        }

        private void sendToDeadLetterQueue(OrderEvent order, Throwable ex) {
            logger.warn("Sending order to dead letter queue: orderId={}",
order.getOrderId());

            // Send to DLQ with error information
            DeadLetterEvent dlqEvent = DeadLetterEvent.builder()
                .originalEvent(order)
                .error(ex.getMessage())
                .timestamp(Instant.now())
                .attempts(1)
                .build();

            kafkaTemplate.send("orders.DLT", order.getOrderId(), dlqEvent);
        }

        private boolean isCriticalOrder(OrderEvent order) {
            // Determine if order is critical (high value, VIP customer, etc.)
            return order.getAmount().compareTo(new java.math.BigDecimal("1000")) >
0;
        }

        private void sendCriticalOrderAlert(OrderEvent order, Throwable ex) {
            logger.error("🚨 CRITICAL ORDER FAILED: orderId={}, amount={}, error=
{}",
                order.getOrderId(), order.getAmount(), ex.getMessage());

            // Send alert to operations team
            // Could use email, Slack, PagerDuty, etc.
        }
    }

    /**
     * Service method using custom callbacks
     */
    public void sendOrderWithCustomCallback(OrderEvent order) {
        String correlationId = UUID.randomUUID().toString();

        logger.info("Sending order with custom callback: orderId={},
correlationId={}",
            order.getOrderId(), correlationId);

        // Create message with headers
        Message<OrderEvent> message = MessageBuilder
            .withPayload(order)
```

```
                .setHeader(KafkaHeaders.TOPIC, "orders")
                .setHeader(KafkaHeaders.KEY, order.getOrderId())
                .setHeader("correlation-id", correlationId)
                .setHeader("order-amount", order.getAmount().toString())
                .setHeader("customer-id", order.getCustomerId())
                .setHeader("timestamp", order.getTimestamp().toEpochMilli())
                .build();

        // Send with custom callback
        CompletableFuture<SendResult<String, Object>> future =
kafkaTemplate.send(message);

        // Register custom callback
        future.whenComplete((result, ex) -> {
            if (ex != null) {
                new OrderEventCallback(order, correlationId).onFailure(ex);
            } else {
                new OrderEventCallback(order, correlationId).onSuccess(result);
            }
        });
    }

    /**
     * Batch sending with individual callbacks
     */
    public void sendOrderBatchWithCallbacks(java.util.List<OrderEvent> orders) {
        logger.info("Sending batch of {} orders with individual callbacks",
orders.size());

        for (OrderEvent order : orders) {
            sendOrderWithCustomCallback(order);
        }

        logger.info("Batch send initiated with {} individual callbacks",
orders.size());
    }

    /**
     * Composite callback for complex scenarios
     */
    public class CompositeCallback implements
ListenableFutureCallback<SendResult<String, Object>> {

        private final java.util.List<ListenableFutureCallback<SendResult<String,
Object>>> callbacks;

        public
CompositeCallback(java.util.List<ListenableFutureCallback<SendResult<String,
Object>>> callbacks) {
            this.callbacks = callbacks;
        }

        @Override
        public void onSuccess(SendResult<String, Object> result) {
```

```
            for (ListenableFutureCallback<SendResult<String, Object>> callback :
callbacks) {
                try {
                    callback.onSuccess(result);
                } catch (Exception e) {
                    logger.error("Error in composite callback success handler",
e);
                }
            }
        }

        @Override
        public void onFailure(Throwable ex) {
            for (ListenableFutureCallback<SendResult<String, Object>> callback :
callbacks) {
                try {
                    callback.onFailure(ex);
                } catch (Exception e) {
                    logger.error("Error in composite callback failure handler",
e);
                }
            }
        }
    }

    /**
     * Circuit breaker pattern with callbacks
     */
    public void sendWithCircuitBreaker(String topic, String key, Object message) {
        if (isCircuitOpen(topic)) {
            logger.warn("Circuit breaker is OPEN for topic: {}", topic);
            handleCircuitOpenFailure(topic, key, message);
            return;
        }

        CompletableFuture<SendResult<String, Object>> future =
kafkaTemplate.send(topic, key, message);

        future.whenComplete((result, ex) -> {
            if (ex != null) {
                recordCircuitBreakerFailure(topic);
            } else {
                recordCircuitBreakerSuccess(topic);
            }
        });
    }

    // Helper methods for metrics and circuit breaker
    private void updateTopicMetrics(String topic, long duration) {
        topicMetrics.put(topic, duration);
    }

    private boolean isCircuitOpen(String topic) {
        // Implement circuit breaker logic
```

```
            return false; // Simplified
        }

        private void handleCircuitOpenFailure(String topic, String key, Object
message) {
            // Handle circuit breaker open state
            logger.warn("Handling circuit breaker open state for topic: {}", topic);
        }

        private void recordCircuitBreakerFailure(String topic) {
            // Record failure for circuit breaker
            logger.debug("Recording circuit breaker failure for topic: {}", topic);
        }

        private void recordCircuitBreakerSuccess(String topic) {
            // Record success for circuit breaker
            logger.debug("Recording circuit breaker success for topic: {}", topic);
        }

        // Public methods for metrics
        public long getSuccessCount() { return successCount.get(); }
        public long getFailureCount() { return failureCount.get(); }
        public Map<String, Long> getTopicMetrics() { return new java.util.HashMap<>
(topicMetrics); }
    }

    /**
     * Dead letter queue event structure
     */
    @lombok.Data
    @lombok.Builder
    @lombok.NoArgsConstructor
    @lombok.AllArgsConstructor
    class DeadLetterEvent {
        private OrderEvent originalEvent;
        private String error;
        private Instant timestamp;
        private int attempts;
        private String correlationId;
    }
```

⚙ Producer Configuration

Key/Value Serializers

Comprehensive Serialization Configuration and Custom Implementations

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.kafka.core.*;
```

```
import org.springframework.kafka.support.serializer.JsonSerializer;
import org.springframework.kafka.support.serializer.DelegatingSerializer;
import org.springframework.kafka.support.serializer.DelegatingByTopicSerializer;

import org.apache.kafka.clients.producer.ProducerConfig;
import org.apache.kafka.common.serialization.*;
import org.apache.kafka.common.header.Headers;

import com.fasterxml.jackson.databind.ObjectMapper;
import com.fasterxml.jackson.datatype.jsr310.JavaTimeModule;

import java.util.HashMap;
import java.util.Map;
import java.util.regex.Pattern;

/**
 * Comprehensive serialization configuration for Spring Kafka producers
 */
@Configuration
public class ProducerSerializationConfiguration {

    /**
     * Basic String serialization producer factory
     */
    @Bean
    public ProducerFactory<String, String> stringProducerFactory() {
        Map<String, Object> configProps = new HashMap<>();

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

        return new DefaultKafkaProducerFactory<>(configProps);
    }

    /**
     * JSON serialization producer factory with custom ObjectMapper
     */
    @Bean
    public ProducerFactory<String, Object> jsonProducerFactory() {
        Map<String, Object> configProps = new HashMap<>();

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

        // JSON serializer specific configuration
```

```
        configProps.put(JsonSerializer.ADD_TYPE_INFO_HEADERS, true);
        configProps.put(JsonSerializer.TYPE_MAPPINGS,
            "order:com.example.OrderEvent," +
            "notification:com.example.NotificationEvent," +
            "payment:com.example.PaymentEvent");

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

        // Custom ObjectMapper for JSON serialization
        ObjectMapper objectMapper = new ObjectMapper();
        objectMapper.registerModule(new JavaTimeModule());

objectMapper.configure(com.fasterxml.jackson.databind.SerializationFeature.WRITE_D
ATES_AS_TIMESTAMPS, false);

        // Set custom ObjectMapper
        JsonSerializer<Object> jsonSerializer = new JsonSerializer<>
(objectMapper);
        factory.setValueSerializer(jsonSerializer);

        return factory;
    }

    /**
     * Avro serialization producer factory (requires Schema Registry)
     */
    @Bean
    public ProducerFactory<String, Object> avroProducerFactory() {
        Map<String, Object> configProps = new HashMap<>();

        configProps.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,
"localhost:9092");
        configProps.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
StringSerializer.class);
        configProps.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
            "io.confluent.kafka.serializers.KafkaAvroSerializer");

        // Schema Registry configuration
        configProps.put("schema.registry.url", "http://localhost:8081");
        configProps.put("auto.register.schemas", true);
        configProps.put("use.latest.version", true);

        return new DefaultKafkaProducerFactory<>(configProps);
    }

    /**
     * Multi-type serialization using DelegatingSerializer
     */
    @Bean
    public ProducerFactory<String, Object> multiTypeProducerFactory() {
        Map<String, Object> configProps = new HashMap<>();

        configProps.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,
```

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

        // Configure delegating serializer mappings
        Map<String, Serializer<?>> delegates = new HashMap<>();
        delegates.put("string", new StringSerializer());
        delegates.put("json", new JsonSerializer<>());
        delegates.put("byte", new ByteArraySerializer());

        // Delegating serializer configuration
        configProps.put(DelegatingSerializer.VALUE_SERIALIZATION_SELECTOR_CONFIG,
            "string:org.apache.kafka.common.serialization.StringSerializer," +
            "json:org.springframework.kafka.support.serializer.JsonSerializer," +
            "byte:org.apache.kafka.common.serialization.ByteArraySerializer");

        return new DefaultKafkaProducerFactory<>(configProps);
    }

    /**
     * Topic-based serialization using DelegatingByTopicSerializer
     */
    @Bean
    public ProducerFactory<String, Object> topicBasedProducerFactory() {
        Map<String, Object> configProps = new HashMap<>();

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

        // Topic-based serializer configuration

configProps.put(DelegatingByTopicSerializer.VALUE_SERIALIZATION_TOPIC_CONFIG,

"orders.*:org.springframework.kafka.support.serializer.JsonSerializer," +
            "logs.*:org.apache.kafka.common.serialization.StringSerializer," +
            "binary.*:org.apache.kafka.common.serialization.ByteArraySerializer");

        return new DefaultKafkaProducerFactory<>(configProps);
    }

    /**
     * Custom serializer implementation
     */
    public static class CustomOrderEventSerializer implements
Serializer<OrderEvent> {

        private final ObjectMapper objectMapper;
```

```
        public CustomOrderEventSerializer() {
            this.objectMapper = new ObjectMapper();
            this.objectMapper.registerModule(new JavaTimeModule());
        }

        @Override
        public void configure(Map<String, ?> configs, boolean isKey) {
            // Configuration if needed
        }

        @Override
        public byte[] serialize(String topic, OrderEvent data) {
            if (data == null) {
                return null;
            }

            try {
                // Add custom serialization logic
                Map<String, Object> customFormat = new HashMap<>();
                customFormat.put("order_id", data.getOrderId());
                customFormat.put("customer_id", data.getCustomerId());
                customFormat.put("amount", data.getAmount().toString());
                customFormat.put("status", data.getStatus());
                customFormat.put("timestamp", data.getTimestamp().toEpochMilli());
                customFormat.put("serialization_version", "1.0");
                customFormat.put("topic", topic);

                return objectMapper.writeValueAsBytes(customFormat);

            } catch (Exception e) {
                throw new RuntimeException("Error serializing OrderEvent", e);
            }
        }

        @Override
        public byte[] serialize(String topic, Headers headers, OrderEvent data) {
            // Add headers information to serialization if needed
            headers.add("serializer", "CustomOrderEventSerializer".getBytes());
            headers.add("version", "1.0".getBytes());

            return serialize(topic, data);
        }

        @Override
        public void close() {
            // Cleanup if needed
        }
    }

    /**
     * Header-aware serializer that includes metadata
     */
    public static class HeaderAwareSerializer implements Serializer<Object> {
```

```
        private final JsonSerializer<Object> jsonSerializer;

        public HeaderAwareSerializer() {
            this.jsonSerializer = new JsonSerializer<>();
        }

        @Override
        public void configure(Map<String, ?> configs, boolean isKey) {
            jsonSerializer.configure(configs, isKey);
        }

        @Override
        public byte[] serialize(String topic, Headers headers, Object data) {
            // Add serialization metadata to headers
            headers.add("content-type", "application/json".getBytes());
            headers.add("serialization-timestamp",
                String.valueOf(System.currentTimeMillis()).getBytes());
            headers.add("data-class", data.getClass().getName().getBytes());

            // Add topic-specific headers
            if (topic.startsWith("orders")) {
                headers.add("business-domain", "order-management".getBytes());
            } else if (topic.startsWith("payments")) {
                headers.add("business-domain", "payment-processing".getBytes());
            }

            return jsonSerializer.serialize(topic, headers, data);
        }

        @Override
        public byte[] serialize(String topic, Object data) {
            return jsonSerializer.serialize(topic, data);
        }

        @Override
        public void close() {
            jsonSerializer.close();
        }
    }

    /**
     * Encryption-aware serializer for sensitive data
     */
    public static class EncryptingSerializer implements Serializer<Object> {

        private final JsonSerializer<Object> delegateSerializer;
        private final EncryptionService encryptionService;

        public EncryptingSerializer(EncryptionService encryptionService) {
            this.delegateSerializer = new JsonSerializer<>();
            this.encryptionService = encryptionService;
        }

        @Override
```

```
        public void configure(Map<String, ?> configs, boolean isKey) {
            delegateSerializer.configure(configs, isKey);
        }

        @Override
        public byte[] serialize(String topic, Object data) {
            // Serialize first
            byte[] serializedData = delegateSerializer.serialize(topic, data);

            // Encrypt if topic contains sensitive data
            if (isSensitiveTopic(topic)) {
                return encryptionService.encrypt(serializedData);
            }

            return serializedData;
        }

        @Override
        public byte[] serialize(String topic, Headers headers, Object data) {
            // Add encryption headers
            if (isSensitiveTopic(topic)) {
                headers.add("encrypted", "true".getBytes());
                headers.add("encryption-algorithm", "AES-256-GCM".getBytes());
            }

            return serialize(topic, data);
        }

        private boolean isSensitiveTopic(String topic) {
            return topic.contains("payment") ||
                   topic.contains("personal") ||
                   topic.contains("sensitive");
        }

        @Override
        public void close() {
            delegateSerializer.close();
        }
    }

    /**
     * KafkaTemplate beans for different serialization strategies
     */
    @Bean
    public KafkaTemplate<String, String> stringKafkaTemplate() {
        return new KafkaTemplate<>(stringProducerFactory());
    }

    @Bean
    public KafkaTemplate<String, Object> jsonKafkaTemplate() {
        return new KafkaTemplate<>(jsonProducerFactory());
    }

    @Bean
```

```
    public KafkaTemplate<String, Object> avroKafkaTemplate() {
        return new KafkaTemplate<>(avroProducerFactory());
    }

    @Bean
    public KafkaTemplate<String, Object> multiTypeKafkaTemplate() {
        return new KafkaTemplate<>(multiTypeProducerFactory());
    }

    /**
     * Service demonstrating different serialization patterns
     */
    @Service
    public static class SerializationDemoService {

        @Autowired
        private KafkaTemplate<String, String> stringTemplate;

        @Autowired
        private KafkaTemplate<String, Object> jsonTemplate;

        @Autowired
        private KafkaTemplate<String, Object> multiTypeTemplate;

        /**
         * Send string message
         */
        public void sendString(String topic, String key, String message) {
            stringTemplate.send(topic, key, message);
        }

        /**
         * Send JSON object
         */
        public void sendJson(String topic, String key, Object obj) {
            jsonTemplate.send(topic, key, obj);
        }

        /**
         * Send with type-specific serialization
         */
        public void sendWithTypeSelection(String topic, String key, Object obj,
String serializationType) {
            // Set serialization selector header
            Message<Object> message = MessageBuilder
                .withPayload(obj)
                .setHeader(KafkaHeaders.TOPIC, topic)
                .setHeader(KafkaHeaders.KEY, key)
                .setHeader(DelegatingSerializer.VALUE_SERIALIZATION_SELECTOR,
serializationType)
                .build();

            multiTypeTemplate.send(message);
        }
```

```
    }

    // Placeholder for encryption service
    interface EncryptionService {
        byte[] encrypt(byte[] data);
        byte[] decrypt(byte[] encryptedData);
    }
}
```

Custom Partitioners

Advanced Partitioning Strategies and Implementations

```
import org.apache.kafka.clients.producer.Partitioner;
import org.apache.kafka.common.Cluster;
import org.apache.kafka.common.PartitionInfo;
import org.apache.kafka.common.utils.Utils;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

import java.util.Map;
import java.util.List;
import java.util.concurrent.ConcurrentHashMap;
import java.util.concurrent.atomic.AtomicInteger;
import java.util.concurrent.ThreadLocalRandom;

/**
 * Custom partitioner implementations for different business requirements
 */
@Configuration
public class CustomPartitionerConfiguration {

    /**
     * Producer factory with custom partitioner
     */
    @Bean
    public ProducerFactory<String, Object> customPartitionerProducerFactory() {
        Map<String, Object> configProps = new HashMap<>();

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

        // Set custom partitioner
        configProps.put(ProducerConfig.PARTITIONER_CLASS_CONFIG,
BusinessLogicPartitioner.class.getName());

        // Custom partitioner properties
```

```
        configProps.put("partitioner.business.rule", "customer-hash");
        configProps.put("partitioner.vip.partition", "0");

        return new DefaultKafkaProducerFactory<>(configProps);
    }

    /**
     * Business logic-based partitioner
     * Partitions messages based on business rules rather than just key hash
     */
    public static class BusinessLogicPartitioner implements Partitioner {

        private static final Logger logger =
LoggerFactory.getLogger(BusinessLogicPartitioner.class);

        private String businessRule;
        private int vipPartition;
        private final AtomicInteger counter = new AtomicInteger(0);

        @Override
        public void configure(Map<String, ?> configs) {
            this.businessRule = (String)
configs.getOrDefault("partitioner.business.rule", "round-robin");
            this.vipPartition = Integer.parseInt(
                configs.getOrDefault("partitioner.vip.partition",
"0").toString());

            logger.info("BusinessLogicPartitioner configured: rule={},
vipPartition={}",
                businessRule, vipPartition);
        }

        @Override
        public int partition(String topic, Object key, byte[] keyBytes,
                             Object value, byte[] valueBytes, Cluster cluster) {

            List<PartitionInfo> partitions = cluster.partitionsForTopic(topic);
            int numPartitions = partitions.size();

            if (numPartitions == 1) {
                return 0;
            }

            // Business-specific partitioning logic
            if (value instanceof OrderEvent) {
                return partitionOrderEvent((OrderEvent) value, numPartitions);
            } else if (value instanceof PaymentEvent) {
                return partitionPaymentEvent((PaymentEvent) value, numPartitions);
            } else if (key != null) {
                return partitionByKey(key.toString(), numPartitions);
            } else {
                return partitionRoundRobin(numPartitions);
            }
        }
```

```
        private int partitionOrderEvent(OrderEvent order, int numPartitions) {
            // VIP customers go to dedicated partition for priority processing
            if (isVipCustomer(order.getCustomerId())) {
                logger.debug("Routing VIP customer {} to partition {}",
                    order.getCustomerId(), vipPartition);
                return vipPartition % numPartitions;
            }

            // High-value orders go to specific partitions
            if (order.getAmount().compareTo(new java.math.BigDecimal("1000")) > 0)
{
                // Use last partition for high-value orders
                int highValuePartition = numPartitions - 1;
                logger.debug("Routing high-value order {} to partition {}",
                    order.getOrderId(), highValuePartition);
                return highValuePartition;
            }

            // Regular orders: hash by customer ID for ordering per customer
            return Math.abs(order.getCustomerId().hashCode()) % numPartitions;
        }

        private int partitionPaymentEvent(PaymentEvent payment, int numPartitions)
{
            // Different partitioning strategy for payments
            // Route by payment method for specialized processing
            switch (payment.getPaymentMethod().toUpperCase()) {
                case "CREDIT_CARD":
                    return 0 % numPartitions;
                case "BANK_TRANSFER":
                    return 1 % numPartitions;
                case "DIGITAL_WALLET":
                    return 2 % numPartitions;
                default:
                    return Math.abs(payment.getPaymentId().hashCode()) %
numPartitions;
            }
        }

        private int partitionByKey(String key, int numPartitions) {
            // Custom key-based partitioning
            switch (businessRule) {
                case "customer-hash":
                    // Extract customer ID from key if present
                    if (key.startsWith("CUSTOMER-")) {
                        String customerId = key.substring(9);
                        return Math.abs(customerId.hashCode()) % numPartitions;
                    }
                    break;
                case "region-based":
                    // Route by geographical region
                    if (key.contains("-US-")) return 0 % numPartitions;
                    if (key.contains("-EU-")) return 1 % numPartitions;
```

```
                    if (key.contains("-ASIA-")) return 2 % numPartitions;
                    break;
            }

            // Default to hash-based partitioning
            return Math.abs(key.hashCode()) % numPartitions;
        }

        private int partitionRoundRobin(int numPartitions) {
            return counter.getAndIncrement() % numPartitions;
        }

        private boolean isVipCustomer(String customerId) {
            // Business logic to determine VIP status
            // Could check against database, cache, or customer service
            return customerId.startsWith("VIP-") || customerId.endsWith("-
PREMIUM");
        }

        @Override
        public void close() {
            logger.info("BusinessLogicPartitioner closed");
        }
    }

    /**
     * Sticky partitioner with custom sticky logic
     * Improves batching efficiency while maintaining business rules
     */
    public static class CustomStickyPartitioner implements Partitioner {

        private final Map<String, Integer> topicPartitionMap = new
ConcurrentHashMap<>();
        private final AtomicInteger batchCounter = new AtomicInteger(0);

        // Switch partition after this many messages to balance load
        private static final int STICKY_BATCH_SIZE = 100;

        @Override
        public void configure(Map<String, ?> configs) {
            // Configuration if needed
        }

        @Override
        public int partition(String topic, Object key, byte[] keyBytes,
                             Object value, byte[] valueBytes, Cluster cluster) {

            List<PartitionInfo> partitions = cluster.partitionsForTopic(topic);
            int numPartitions = partitions.size();

            if (numPartitions == 1) {
                return 0;
            }
```

```
            // If key is provided, use key-based partitioning for ordering
            if (key != null) {
                return Math.abs(key.hashCode()) % numPartitions;
            }

            // Sticky logic for keyless messages
            String topicKey = topic;
            int currentBatch = batchCounter.incrementAndGet();

            // Check if we should switch partition
            if (currentBatch % STICKY_BATCH_SIZE == 0) {
                // Switch to next partition
                int currentPartition = topicPartitionMap.getOrDefault(topicKey,
-1);
                int nextPartition = (currentPartition + 1) % numPartitions;
                topicPartitionMap.put(topicKey, nextPartition);

                logger.debug("Switching sticky partition for topic {} to partition
{}",
                    topic, nextPartition);
                return nextPartition;
            }

            // Use current sticky partition
            return topicPartitionMap.computeIfAbsent(topicKey,
                k -> ThreadLocalRandom.current().nextInt(numPartitions));
        }

        @Override
        public void close() {
            topicPartitionMap.clear();
        }
    }

    /**
     * Load-aware partitioner that considers partition load
     */
    public static class LoadAwarePartitioner implements Partitioner {

        private final Map<String, PartitionLoadTracker> loadTrackers = new
ConcurrentHashMap<>();

        @Override
        public void configure(Map<String, ?> configs) {
            // Configuration if needed
        }

        @Override
        public int partition(String topic, Object key, byte[] keyBytes,
                             Object value, byte[] valueBytes, Cluster cluster) {

            List<PartitionInfo> partitions = cluster.partitionsForTopic(topic);
            int numPartitions = partitions.size();
```

```
            if (numPartitions == 1) {
                return 0;
            }

            // Get or create load tracker for topic
            PartitionLoadTracker tracker = loadTrackers.computeIfAbsent(topic,
                k -> new PartitionLoadTracker(numPartitions));

            // If key is provided and ordering is important, use key-based
partitioning
            if (key != null && requiresOrdering(value)) {
                return Math.abs(key.hashCode()) % numPartitions;
            }

            // Choose least loaded partition
            int selectedPartition = tracker.getLeastLoadedPartition();
            tracker.recordMessage(selectedPartition);

            logger.debug("Selected partition {} for topic {} (load-based)",
selectedPartition, topic);
            return selectedPartition;
        }

        private boolean requiresOrdering(Object value) {
            // Determine if message type requires ordering
            return value instanceof OrderEvent ||
                   value instanceof PaymentEvent ||
                   (value instanceof Map && ((Map<?, ?>)
value).containsKey("requiresOrdering"));
        }

        @Override
        public void close() {
            loadTrackers.clear();
        }

        /**
         * Simple load tracker for partitions
         */
        private static class PartitionLoadTracker {
            private final AtomicInteger[] partitionCounts;
            private final int numPartitions;

            public PartitionLoadTracker(int numPartitions) {
                this.numPartitions = numPartitions;
                this.partitionCounts = new AtomicInteger[numPartitions];
                for (int i = 0; i < numPartitions; i++) {
                    this.partitionCounts[i] = new AtomicInteger(0);
                }
            }

            public int getLeastLoadedPartition() {
                int minLoad = Integer.MAX_VALUE;
                int selectedPartition = 0;
```

```
                for (int i = 0; i < numPartitions; i++) {
                    int currentLoad = partitionCounts[i].get();
                    if (currentLoad < minLoad) {
                        minLoad = currentLoad;
                        selectedPartition = i;
                    }
                }

                return selectedPartition;
            }

            public void recordMessage(int partition) {
                if (partition >= 0 && partition < numPartitions) {
                    partitionCounts[partition].incrementAndGet();

                    // Decay counters periodically to adapt to changing load
patterns
                    if (partitionCounts[partition].get() % 1000 == 0) {
                        for (AtomicInteger counter : partitionCounts) {
                            counter.updateAndGet(val -> val / 2); // Simple decay
                        }
                    }
                }
            }
        }
    }

    /**
     * Time-based partitioner for time-series data
     */
    public static class TimeBasedPartitioner implements Partitioner {

        private long partitionTimeWindowMs;

        @Override
        public void configure(Map<String, ?> configs) {
            // Default to 1 hour time windows
            this.partitionTimeWindowMs = Long.parseLong(
                configs.getOrDefault("partitioner.time.window.ms",
"3600000").toString());
        }

        @Override
        public int partition(String topic, Object key, byte[] keyBytes,
                             Object value, byte[] valueBytes, Cluster cluster) {

            List<PartitionInfo> partitions = cluster.partitionsForTopic(topic);
            int numPartitions = partitions.size();

            if (numPartitions == 1) {
                return 0;
            }
```

```
            // Get timestamp from message
            long timestamp = extractTimestamp(value);

            // Calculate time-based partition
            long timeWindow = timestamp / partitionTimeWindowMs;
            int timePartition = (int) (timeWindow % numPartitions);

            logger.debug("Time-based partition {} selected for timestamp {}",
timePartition, timestamp);
            return timePartition;
        }

        private long extractTimestamp(Object value) {
            if (value instanceof OrderEvent) {
                return ((OrderEvent) value).getTimestamp().toEpochMilli();
            } else if (value instanceof Map) {
                Map<?, ?> map = (Map<?, ?>) value;
                if (map.containsKey("timestamp")) {
                    return Long.parseLong(map.get("timestamp").toString());
                }
            }

            // Default to current time
            return System.currentTimeMillis();
        }

        @Override
        public void close() {
            // Cleanup if needed
        }
    }

    /**
     * Service demonstrating custom partitioner usage
     */
    @Service
    public static class PartitioningDemoService {

        @Autowired
        private KafkaTemplate<String, Object> kafkaTemplate;

        /**
         * Send message with explicit partition
         */
        public void sendToSpecificPartition(String topic, String key, Object
message, int partition) {
            kafkaTemplate.send(topic, partition, key, message);
        }

        /**
         * Send message letting custom partitioner decide
         */
        public void sendWithCustomPartitioning(String topic, String key, Object
message) {
```

```
            // Custom partitioner will be used automatically
            kafkaTemplate.send(topic, key, message);
        }

        /**
         * Send VIP customer order (will be routed to VIP partition)
         */
        public void sendVipOrder(OrderEvent order) {
            kafkaTemplate.send("orders", order.getCustomerId(), order);
        }

        /**
         * Send regular order (will use customer-based partitioning)
         */
        public void sendRegularOrder(OrderEvent order) {
            kafkaTemplate.send("orders", order.getCustomerId(), order);
        }
    }
}

/**
 * Payment event for partitioner examples
 */
@lombok.Data
@lombok.Builder
@lombok.NoArgsConstructor
@lombok.AllArgsConstructor
class PaymentEvent {
    private String paymentId;
    private String orderId;
    private java.math.BigDecimal amount;
    private String paymentMethod; // CREDIT_CARD, BANK_TRANSFER, DIGITAL_WALLET
    private String status;
    private Instant timestamp;
}
```

Transactional Producers

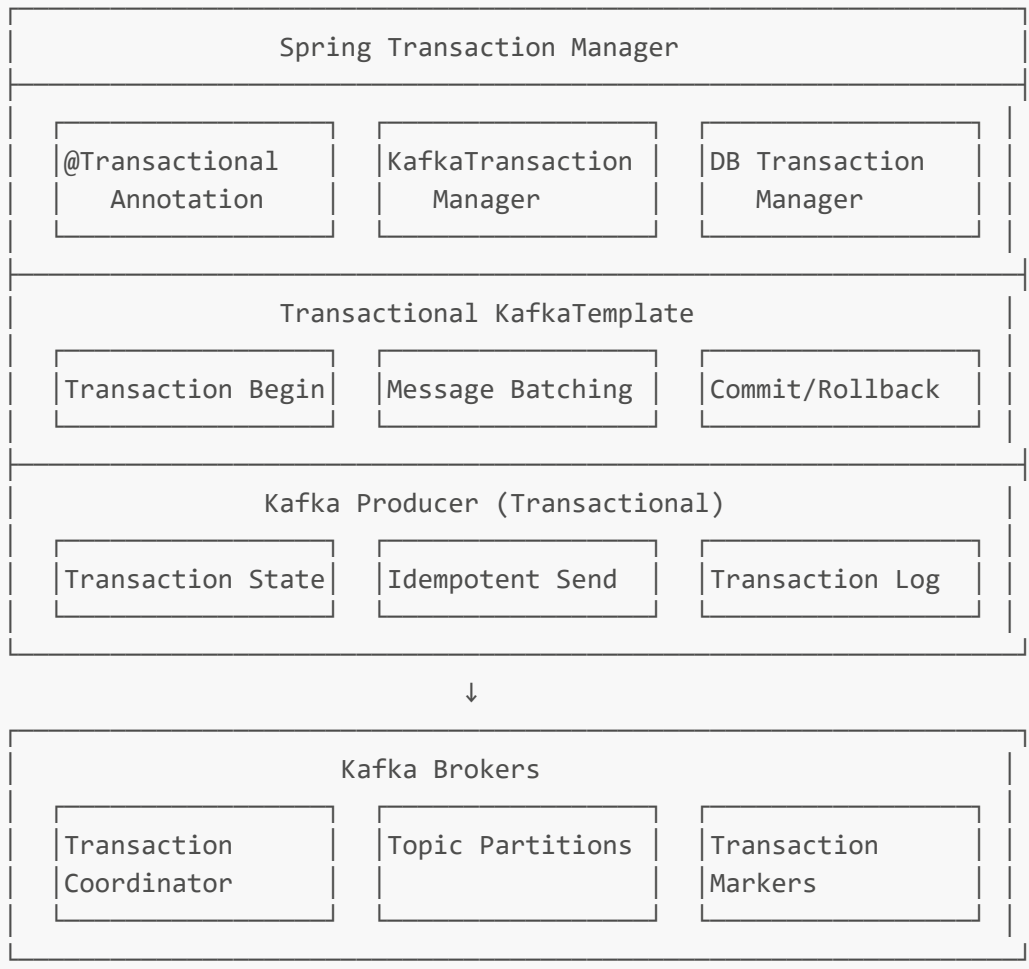
Enabling Transactions

Simple Explanation: Transactional producers in Spring Kafka provide exactly-once semantics by ensuring that a group of messages are either all successfully delivered or none at all, maintaining data consistency across multiple topics and partitions.

Problem It Solves:

- **Atomicity**: Ensures all messages in a transaction succeed or fail together
- **Consistency**: Maintains data integrity across multiple topics
- **Duplicate Prevention**: Prevents message duplication during retries
- **Cross-System Coordination**: Coordinates Kafka operations with database transactions

Internal Architecture:



Comprehensive Transactional Producer Configuration

```
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.kafka.core.*;
import org.springframework.kafka.transaction.KafkaTransactionManager;
import org.springframework.transaction.annotation.EnableTransactionManagement;
import org.springframework.transaction.PlatformTransactionManager;

import org.apache.kafka.clients.producer.ProducerConfig;
import org.apache.kafka.common.serialization.StringSerializer;
import org.springframework.kafka.support.serializer.JsonSerializer;

import java.util.HashMap;
import java.util.Map;

/**
 * Comprehensive transactional producer configuration
```

```
 */
@Configuration
@EnableTransactionManagement
public class TransactionalProducerConfiguration {

    @Value("${spring.kafka.bootstrap-servers:localhost:9092}")
    private String bootstrapServers;

    /**
     * Transactional producer factory configuration
     */
    @Bean
    public ProducerFactory<String, Object> transactionalProducerFactory() {
        Map<String, Object> configProps = new HashMap<>();

        // Basic Kafka configuration
        configProps.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,
bootstrapServers);
        configProps.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
StringSerializer.class);
        configProps.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
JsonSerializer.class);

        // Transaction configuration
        configProps.put(ProducerConfig.TRANSACTIONAL_ID_CONFIG, "tx-producer");
        configProps.put(ProducerConfig.ENABLE_IDEMPOTENCE_CONFIG, true);

        // Reliability settings (required for transactions)
        configProps.put(ProducerConfig.ACKS_CONFIG, "all");
        configProps.put(ProducerConfig.RETRIES_CONFIG, Integer.MAX_VALUE);
        configProps.put(ProducerConfig.MAX_IN_FLIGHT_REQUESTS_PER_CONNECTION, 1);

        // Performance settings
        configProps.put(ProducerConfig.BATCH_SIZE_CONFIG, 16384);
        configProps.put(ProducerConfig.LINGER_MS_CONFIG, 5);
        configProps.put(ProducerConfig.BUFFER_MEMORY_CONFIG, 33554432);

        // Transaction timeout
        configProps.put(ProducerConfig.TRANSACTION_TIMEOUT_CONFIG, 60000); // 60
seconds

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

        // Set transaction ID prefix for multiple producer instances
        factory.setTransactionIdPrefix("tx-producer-");

        return factory;
    }

    /**
     * Transactional KafkaTemplate
     */
    @Bean
```

```
    public KafkaTemplate<String, Object> transactionalKafkaTemplate() {
        KafkaTemplate<String, Object> template = new KafkaTemplate<>
(transactionalProducerFactory());

        // Set producer listener for monitoring
        template.setProducerListener(transactionalProducerListener());

        return template;
    }

    /**
     * Kafka Transaction Manager
     */
    @Bean("kafkaTransactionManager")
    public KafkaTransactionManager kafkaTransactionManager() {
        return new KafkaTransactionManager(transactionalProducerFactory());
    }

    /**
     * Multi-producer factory for high-throughput scenarios
     */
    @Bean
    public ProducerFactory<String, Object>
multiInstanceTransactionalProducerFactory() {
        Map<String, Object> configProps = new HashMap<>();

        // Copy base configuration
        configProps.putAll(((DefaultKafkaProducerFactory<String, Object>)
            transactionalProducerFactory()).getConfigurationProperties());

        // Different transaction ID for different instance
        configProps.put(ProducerConfig.TRANSACTIONAL_ID_CONFIG, "tx-producer-
multi");

        DefaultKafkaProducerFactory<String, Object> factory = new
DefaultKafkaProducerFactory<>(configProps);
        factory.setTransactionIdPrefix("tx-multi-");

        return factory;
    }

    /**
     * Producer listener for transaction monitoring
     */
    @Bean
    public ProducerListener<String, Object> transactionalProducerListener() {
        return new ProducerListener<String, Object>() {
            @Override
            public void onSuccess(ProducerRecord<String, Object> producerRecord,
                                  RecordMetadata recordMetadata) {
                logger.debug("Transactional message sent: topic={}, partition={},
offset={}",
                    recordMetadata.topic(), recordMetadata.partition(),
recordMetadata.offset());
```

```
            }

            @Override
            public void onError(ProducerRecord<String, Object> producerRecord,
                                RecordMetadata recordMetadata, Exception exception)
{
                logger.error("Transactional message failed: topic={}, key={},
error={}",
                    producerRecord.topic(), producerRecord.key(),
exception.getMessage());
            }
        };
    }
}
```

Advanced Transactional Service Implementation

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.kafka.core.KafkaTemplate;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;
import org.springframework.transaction.support.TransactionTemplate;
import org.springframework.transaction.PlatformTransactionManager;

import javax.sql.DataSource;
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.SQLException;
import java.util.List;
import java.util.concurrent.CompletableFuture;

/**
 * Comprehensive transactional producer service
 */
@Service
public class TransactionalProducerService {

    private static final Logger logger =
LoggerFactory.getLogger(TransactionalProducerService.class);

    @Autowired
    private KafkaTemplate<String, Object> transactionalKafkaTemplate;

    @Autowired
    private PlatformTransactionManager kafkaTransactionManager;

    @Autowired
    private PlatformTransactionManager dbTransactionManager;

    @Autowired
    private DataSource dataSource;
```

```
    // ======================================
    // DECLARATIVE TRANSACTION PATTERNS
    // ======================================

    /**
     * Simple transactional send - all messages succeed or fail together
     */
    @Transactional("kafkaTransactionManager")
    public void sendTransactionalMessages(List<OrderEvent> orders) {
        logger.info("Sending {} orders in transaction", orders.size());

        try {
            for (OrderEvent order : orders) {
                // Send order event
                transactionalKafkaTemplate.send("orders", order.getOrderId(),
order);

                // Send related notification
                NotificationEvent notification =
createNotificationFromOrder(order);
                transactionalKafkaTemplate.send("notifications",
order.getCustomerId(), notification);

                // Send audit event
                AuditEvent audit = createAuditEvent(order);
                transactionalKafkaTemplate.send("audit-events",
order.getOrderId(), audit);
            }

            logger.info("All {} orders sent successfully in transaction",
orders.size());

        } catch (Exception e) {
            logger.error("Transaction failed, rolling back all {} orders",
orders.size(), e);
            throw e; // Will trigger rollback
        }
    }

    /**
     * Cross-system transaction: Kafka + Database
     * Note: This requires careful coordination as Kafka transactions
     * are not automatically coordinated with database transactions
     */
    @Transactional("kafkaTransactionManager")
    public void processOrderWithDatabaseUpdate(OrderEvent order) {
        logger.info("Processing order with database update: {}",
order.getOrderId());

        try {
            // 1. Send Kafka messages first
            transactionalKafkaTemplate.send("orders", order.getOrderId(), order);
```

```
            NotificationEvent notification = createNotificationFromOrder(order);
            transactionalKafkaTemplate.send("notifications",
order.getCustomerId(), notification);

            // 2. Database operations (separate transaction)
            updateOrderInDatabase(order);

            logger.info("Order processed successfully: {}", order.getOrderId());

        } catch (Exception e) {
            logger.error("Order processing failed: {}", order.getOrderId(), e);
            throw e;
        }
    }

    /**
     * Compensating transaction pattern for cross-system coordination
     */
    @Transactional("kafkaTransactionManager")
    public void processOrderWithCompensation(OrderEvent order) {
        logger.info("Processing order with compensation pattern: {}",
order.getOrderId());

        boolean kafkaSuccess = false;
        boolean dbSuccess = false;

        try {
            // 1. Kafka transaction
            transactionalKafkaTemplate.send("orders", order.getOrderId(), order);
            kafkaSuccess = true;

            // 2. Database operation (separate transaction)
            dbSuccess = updateOrderInDatabaseSafely(order);

            if (!dbSuccess) {
                // Send compensation message
                OrderCancellationEvent cancellation =
OrderCancellationEvent.builder()
                    .orderId(order.getOrderId())
                    .reason("Database update failed")
                    .originalOrder(order)
                    .build();

                transactionalKafkaTemplate.send("order-cancellations",
order.getOrderId(), cancellation);

                throw new RuntimeException("Database update failed, compensation
sent");
            }

            logger.info("Order processed successfully with compensation pattern:
{}", order.getOrderId());

        } catch (Exception e) {
```

```
            logger.error("Order processing failed: {}", order.getOrderId(), e);
            throw e;
        }
    }

    /**
     * Conditional transactional processing
     */
    @Transactional("kafkaTransactionManager")
    public void processOrderConditionally(OrderEvent order) {
        logger.info("Processing order conditionally: {}", order.getOrderId());

        try {
            // Validate order before processing
            if (!validateOrder(order)) {
                // Send validation failure event
                OrderValidationFailedEvent failureEvent =
OrderValidationFailedEvent.builder()
                    .orderId(order.getOrderId())
                    .validationErrors(getValidationErrors(order))
                    .build();

                transactionalKafkaTemplate.send("order-validation-failed",
                    order.getOrderId(), failureEvent);

                logger.warn("Order validation failed: {}", order.getOrderId());
                return; // Transaction commits with failure event
            }

            // Process valid order
            transactionalKafkaTemplate.send("orders", order.getOrderId(), order);

            // Send success notification
            NotificationEvent notification = createNotificationFromOrder(order);
            transactionalKafkaTemplate.send("notifications",
order.getCustomerId(), notification);

            logger.info("Order processed successfully after validation: {}",
order.getOrderId());

        } catch (Exception e) {
            logger.error("Conditional order processing failed: {}",
order.getOrderId(), e);
            throw e;
        }
    }

    // =======================================
    // PROGRAMMATIC TRANSACTION PATTERNS
    // =======================================

    /**
     * Programmatic transaction with manual control
     */
```

```
    public void sendWithProgrammaticTransaction(List<OrderEvent> orders) {
        logger.info("Sending {} orders with programmatic transaction",
orders.size());

        TransactionTemplate transactionTemplate = new
TransactionTemplate(kafkaTransactionManager);

        transactionTemplate.execute(status -> {
            try {
                for (OrderEvent order : orders) {
                    transactionalKafkaTemplate.send("orders", order.getOrderId(),
order);

                    // Check conditions for rollback
                    if (shouldRollback(order)) {
                        logger.warn("Rolling back transaction due to order: {}",
order.getOrderId());
                        status.setRollbackOnly();
                        return null;
                    }
                }

                logger.info("Programmatic transaction completed successfully for
{} orders", orders.size());
                return null;

            } catch (Exception e) {
                logger.error("Programmatic transaction failed", e);
                status.setRollbackOnly();
                throw new RuntimeException("Transaction failed", e);
            }
        });
    }

    /**
     * Batch processing with transaction boundaries
     */
    public void processBatchWithTransactions(List<OrderEvent> orders, int
batchSize) {
        logger.info("Processing {} orders in batches of {}", orders.size(),
batchSize);

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

            processBatchTransaction(batch, i / batchSize + 1);
        }
    }

    @Transactional("kafkaTransactionManager")
    private void processBatchTransaction(List<OrderEvent> batch, int batchNumber)
{
        logger.info("Processing batch {}: {} orders", batchNumber, batch.size());
```

```
        try {
            for (OrderEvent order : batch) {
                transactionalKafkaTemplate.send("orders", order.getOrderId(),
order);

                // Add batch metadata
                BatchProcessingEvent batchEvent = BatchProcessingEvent.builder()
                    .orderId(order.getOrderId())
                    .batchNumber(batchNumber)
                    .batchSize(batch.size())
                    .timestamp(Instant.now())
                    .build();

                transactionalKafkaTemplate.send("batch-processing",
order.getOrderId(), batchEvent);
            }

            logger.info("Batch {} processed successfully", batchNumber);

        } catch (Exception e) {
            logger.error("Batch {} processing failed", batchNumber, e);
            throw e;
        }
    }

    // ========================================
    // TRANSACTION CALLBACK PATTERNS
    // ========================================

    /**
     * Transaction with success/failure callbacks
     */
    public CompletableFuture<Void> sendWithTransactionCallbacks(OrderEvent order)
{
        logger.info("Sending order with transaction callbacks: {}",
order.getOrderId());

        return transactionalKafkaTemplate.executeInTransaction(template -> {
            try {
                // Send messages within transaction
                template.send("orders", order.getOrderId(), order);

                NotificationEvent notification =
createNotificationFromOrder(order);
                template.send("notifications", order.getCustomerId(),
notification);

                logger.info("Transaction callback: Order sent successfully: {}",
order.getOrderId());
                return CompletableFuture.completedFuture(null);

            } catch (Exception e) {
                logger.error("Transaction callback: Order send failed: {}",
```

```
order.getOrderId(), e);
                return CompletableFuture.failedFuture(e);
            }
        });
    }

    /**
     * Nested transaction handling
     */
    @Transactional("kafkaTransactionManager")
    public void processOrderWithNestedOperations(OrderEvent order) {
        logger.info("Processing order with nested operations: {}",
order.getOrderId());

        try {
            // Main order processing
            transactionalKafkaTemplate.send("orders", order.getOrderId(), order);

            // Nested operation (part of same transaction)
            processNestedInventoryUpdate(order);

            // Another nested operation
            processNestedPayment(order);

            logger.info("Order with nested operations processed: {}",
order.getOrderId());

        } catch (Exception e) {
            logger.error("Nested operation failed for order: {}",
order.getOrderId(), e);
            throw e;
        }
    }

    private void processNestedInventoryUpdate(OrderEvent order) {
        logger.debug("Processing nested inventory update for order: {}",
order.getOrderId());

        InventoryUpdateEvent inventoryUpdate = InventoryUpdateEvent.builder()
            .orderId(order.getOrderId())
            .productId("PRODUCT-123")
            .quantity(-1)
            .operation("RESERVE")
            .build();

        transactionalKafkaTemplate.send("inventory-updates", order.getOrderId(),
inventoryUpdate);
    }

    private void processNestedPayment(OrderEvent order) {
        logger.debug("Processing nested payment for order: {}",
order.getOrderId());

        PaymentEvent payment = PaymentEvent.builder()
```

```
                .paymentId("PAYMENT-" + UUID.randomUUID())
                .orderId(order.getOrderId())
                .amount(order.getAmount())
                .paymentMethod("CREDIT_CARD")
                .status("PENDING")
                .build();

            transactionalKafkaTemplate.send("payments", order.getOrderId(), payment);
        }

        // ========================================
        // HELPER METHODS
        // ========================================

        private NotificationEvent createNotificationFromOrder(OrderEvent order) {
            return NotificationEvent.builder()
                .userId(order.getCustomerId())
                .message("Order " + order.getOrderId() + " has been placed")
                .type("ORDER_PLACED")
                .timestamp(Instant.now())
                .build();
        }

        private AuditEvent createAuditEvent(OrderEvent order) {
            return AuditEvent.builder()
                .entityId(order.getOrderId())
                .entityType("ORDER")
                .action("CREATED")
                .userId(order.getCustomerId())
                .timestamp(Instant.now())
                .build();
        }

        private boolean validateOrder(OrderEvent order) {
            return order.getAmount() != null &&
                   order.getAmount().compareTo(java.math.BigDecimal.ZERO) > 0 &&
                   order.getCustomerId() != null &&
                   !order.getCustomerId().isEmpty();
        }

        private List<String> getValidationErrors(OrderEvent order) {
            List<String> errors = new ArrayList<>();
            if (order.getAmount() == null ||
    order.getAmount().compareTo(java.math.BigDecimal.ZERO) <= 0) {
                errors.add("Invalid amount");
            }
            if (order.getCustomerId() == null || order.getCustomerId().isEmpty()) {
                errors.add("Missing customer ID");
            }
            return errors;
        }

        private boolean shouldRollback(OrderEvent order) {
            // Business logic to determine rollback conditions
```

```
        return order.getAmount().compareTo(new java.math.BigDecimal("10000")) > 0;
// Example: Very high amounts
    }

    private void updateOrderInDatabase(OrderEvent order) {
        // Database update logic
        logger.debug("Updating order in database: {}", order.getOrderId());
    }

    private boolean updateOrderInDatabaseSafely(OrderEvent order) {
        try {
            updateOrderInDatabase(order);
            return true;
        } catch (Exception e) {
            logger.error("Database update failed for order: {}",
order.getOrderId(), e);
            return false;
        }
    }
}

// Additional domain objects for transaction examples
@lombok.Data
@lombok.Builder
@lombok.NoArgsConstructor
@lombok.AllArgsConstructor
class OrderCancellationEvent {
    private String orderId;
    private String reason;
    private OrderEvent originalOrder;
    private Instant timestamp;
}

@lombok.Data
@lombok.Builder
@lombok.NoArgsConstructor
@lombok.AllArgsConstructor
class OrderValidationFailedEvent {
    private String orderId;
    private List<String> validationErrors;
    private Instant timestamp;
}

@lombok.Data
@lombok.Builder
@lombok.NoArgsConstructor
@lombok.AllArgsConstructor
class BatchProcessingEvent {
    private String orderId;
    private int batchNumber;
    private int batchSize;
    private Instant timestamp;
}
```

```
@lombok.Data
@lombok.Builder
@lombok.NoArgsConstructor
@lombok.AllArgsConstructor
class InventoryUpdateEvent {
    private String orderId;
    private String productId;
    private int quantity;
    private String operation; // RESERVE, RELEASE, ADJUST
    private Instant timestamp;
}

@lombok.Data
@lombok.Builder
@lombok.NoArgsConstructor
@lombok.AllArgsConstructor
class NotificationEvent {
    private String userId;
    private String message;
    private String type;
    private Instant timestamp;
}

@lombok.Data
@lombok.Builder
@lombok.NoArgsConstructor
@lombok.AllArgsConstructor
class AuditEvent {
    private String entityId;
    private String entityType;
    private String action;
    private String userId;
    private Instant timestamp;
}
```

Idempotent Producer

Simple Explanation: Idempotent producers ensure that retrying message sends doesn't create duplicate messages, providing exactly-once delivery semantics at the producer level by assigning sequence numbers to messages and deduplicating on the broker side.

Why Idempotence is Required for Transactions:

- Transactions build upon idempotency for reliability
- Prevents duplicates in transaction state messages
- Ensures transaction markers are not duplicated
- Provides foundation for exactly-once semantics

Comprehensive Idempotent Producer Configuration

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.kafka.core.*;

import org.apache.kafka.clients.producer.ProducerConfig;
import org.apache.kafka.common.serialization.StringSerializer;
import org.springframework.kafka.support.serializer.JsonSerializer;

/**
 * Comprehensive idempotent producer configuration and examples
 */
@Configuration
public class IdempotentProducerConfiguration {

    /**
     * Basic idempotent producer factory
     */
    @Bean
    public ProducerFactory<String, Object> idempotentProducerFactory() {
        Map<String, Object> configProps = new HashMap<>();

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

        // Idempotence configuration
        configProps.put(ProducerConfig.ENABLE_IDEMPOTENCE_CONFIG, true);

        // Required settings for idempotence
        configProps.put(ProducerConfig.ACKS_CONFIG, "all");
        configProps.put(ProducerConfig.RETRIES_CONFIG, Integer.MAX_VALUE);
        configProps.put(ProducerConfig.MAX_IN_FLIGHT_REQUESTS_PER_CONNECTION, 1);

        // Performance settings
        configProps.put(ProducerConfig.BATCH_SIZE_CONFIG, 16384);
        configProps.put(ProducerConfig.LINGER_MS_CONFIG, 5);
        configProps.put(ProducerConfig.COMPRESSION_TYPE_CONFIG, "lz4");

        return new DefaultKafkaProducerFactory<>(configProps);
    }

    /**
     * High-throughput idempotent producer (relaxed ordering)
     */
    @Bean
    public ProducerFactory<String, Object>
highThroughputIdempotentProducerFactory() {
        Map<String, Object> configProps = new HashMap<>();
```

```
        // Copy base idempotent configuration
        configProps.putAll(((DefaultKafkaProducerFactory<String, Object>)
            idempotentProducerFactory()).getConfigurationProperties());

        // Relax ordering constraints for better throughput
        configProps.put(ProducerConfig.MAX_IN_FLIGHT_REQUESTS_PER_CONNECTION, 5);

        // Increase batch settings for throughput
        configProps.put(ProducerConfig.BATCH_SIZE_CONFIG, 32768);
        configProps.put(ProducerConfig.LINGER_MS_CONFIG, 10);
        configProps.put(ProducerConfig.BUFFER_MEMORY_CONFIG, 67108864); // 64MB

        return new DefaultKafkaProducerFactory<>(configProps);
    }

    /**
     * Idempotent KafkaTemplate with custom producer ID
     */
    @Bean
    public KafkaTemplate<String, Object> idempotentKafkaTemplate() {
        KafkaTemplate<String, Object> template = new KafkaTemplate<>
(idempotentProducerFactory());

        // Set producer listener to monitor idempotence
        template.setProducerListener(idempotentProducerListener());

        return template;
    }

    /**
     * Producer listener for idempotence monitoring
     */
    @Bean
    public ProducerListener<String, Object> idempotentProducerListener() {
        return new ProducerListener<String, Object>() {

            private final AtomicLong duplicateCount = new AtomicLong(0);
            private final AtomicLong totalCount = new AtomicLong(0);

            @Override
            public void onSuccess(ProducerRecord<String, Object> producerRecord,
                                  RecordMetadata recordMetadata) {

                totalCount.incrementAndGet();

                logger.debug("Idempotent message sent: topic={}, partition={},
offset={}, key={}",
                    recordMetadata.topic(),
                    recordMetadata.partition(),
                    recordMetadata.offset(),
                    producerRecord.key());

                // Log idempotence metrics periodically
                if (totalCount.get() % 1000 == 0) {
```

```
                    logger.info("Idempotent producer metrics: {} total messages,
{} duplicates avoided",
                            totalCount.get(), duplicateCount.get());
                }
            }

            @Override
            public void onError(ProducerRecord<String, Object> producerRecord,
                                RecordMetadata recordMetadata, Exception exception)
{

                logger.error("Idempotent message failed: topic={}, key={}, error=
{}",
                    producerRecord.topic(),
                    producerRecord.key(),
                    exception.getMessage());

                // Check for specific idempotence-related errors
                if (exception instanceof
org.apache.kafka.common.errors.DuplicateSequenceException) {
                    duplicateCount.incrementAndGet();
                    logger.warn("Duplicate sequence detected - idempotence working
correctly");
                } else if (exception instanceof
org.apache.kafka.common.errors.OutOfOrderSequenceException) {
                    logger.error("Out of order sequence - potential idempotence
issue");
                }
            }
        };
    }
}

/**
 * Service demonstrating idempotent producer patterns
 */
@Service
public class IdempotentProducerService {

    private static final Logger logger =
LoggerFactory.getLogger(IdempotentProducerService.class);

    @Autowired
    private KafkaTemplate<String, Object> idempotentKafkaTemplate;

    // Message deduplication tracking
    private final Set<String> processedMessageIds = ConcurrentHashMap.newKeySet();

    /**
     * Send message with application-level idempotence
     * Combines producer idempotence with application-level deduplication
     */
    public void sendIdempotentMessage(String topic, String messageId, Object
message) {
```

```
        logger.info("Sending idempotent message: topic={}, messageId={}", topic,
messageId);

        // Application-level deduplication check
        if (processedMessageIds.contains(messageId)) {
            logger.info("Message already processed (application-level): messageId=
{}", messageId);
            return;
        }

        try {
            // Create message with idempotence headers
            Message<Object> idempotentMessage = MessageBuilder
                .withPayload(message)
                .setHeader(KafkaHeaders.TOPIC, topic)
                .setHeader(KafkaHeaders.KEY, messageId)
                .setHeader("message-id", messageId)
                .setHeader("idempotence-key", generateIdempotenceKey(messageId,
message))
                .setHeader("send-timestamp", System.currentTimeMillis())
                .setHeader("retry-count", 0)
                .build();

            // Send with producer-level idempotence
            CompletableFuture<SendResult<String, Object>> future =
                idempotentKafkaTemplate.send(idempotentMessage);

            future.whenComplete((result, ex) -> {
                if (ex != null) {
                    logger.error("Idempotent send failed: messageId={}, error={}",
                        messageId, ex.getMessage());
                    handleSendFailure(messageId, message, ex);
                } else {
                    logger.info("Idempotent send successful: messageId={}, offset=
{}",
                        messageId, result.getRecordMetadata().offset());

                    // Mark as processed (application-level)
                    processedMessageIds.add(messageId);
                    handleSendSuccess(messageId, result);
                }
            });

        } catch (Exception e) {
            logger.error("Error sending idempotent message: messageId={}",
messageId, e);
            throw new RuntimeException("Idempotent send failed", e);
        }
    }

    /**
     * Retry mechanism with idempotence
     */
    public void sendWithRetryAndIdempotence(String topic, String messageId,
```

```
                                            Object message, int maxRetries) {

        sendWithRetryAttempt(topic, messageId, message, maxRetries, 0);
    }

    private void sendWithRetryAttempt(String topic, String messageId, Object
message,
                                      int maxRetries, int currentAttempt) {

        logger.info("Sending with retry: messageId={}, attempt={}/{}",
            messageId, currentAttempt + 1, maxRetries);

        try {
            Message<Object> retryMessage = MessageBuilder
                .withPayload(message)
                .setHeader(KafkaHeaders.TOPIC, topic)
                .setHeader(KafkaHeaders.KEY, messageId)
                .setHeader("message-id", messageId)
                .setHeader("retry-count", currentAttempt)
                .setHeader("max-retries", maxRetries)
                .setHeader("retry-timestamp", System.currentTimeMillis())
                .build();

            CompletableFuture<SendResult<String, Object>> future =
                idempotentKafkaTemplate.send(retryMessage);

            future.whenComplete((result, ex) -> {
                if (ex != null) {
                    if (currentAttempt < maxRetries) {
                        logger.warn("Retry attempt {} failed, retrying: messageId=
{}",
                            currentAttempt + 1, messageId);

                        // Exponential backoff
                        int delayMs = (int) Math.pow(2, currentAttempt) * 1000;

                        CompletableFuture.delayedExecutor(delayMs,
TimeUnit.MILLISECONDS)
                            .execute(() -> sendWithRetryAttempt(topic, messageId,
message,
                                maxRetries, currentAttempt + 1));
                    } else {
                        logger.error("All retry attempts exhausted: messageId={}",
messageId);
                        handleFinalFailure(messageId, message, ex);
                    }
                } else {
                    logger.info("Retry successful: messageId={}, attempt={},
offset={}",
                        messageId, currentAttempt + 1,
result.getRecordMetadata().offset());
                }
            });
```

```
            } catch (Exception e) {
                logger.error("Error in retry attempt: messageId={}, attempt={}",
                    messageId, currentAttempt + 1, e);

                if (currentAttempt < maxRetries) {
                    sendWithRetryAttempt(topic, messageId, message, maxRetries,
currentAttempt + 1);
                } else {
                    handleFinalFailure(messageId, message, e);
                }
            }
        }

        /**
         * Batch sending with idempotence
         */
        public CompletableFuture<BatchIdempotentResult> sendBatchIdempotent(
                String topic, List<OrderEvent> orders) {

            logger.info("Sending batch of {} orders with idempotence", orders.size());

            List<CompletableFuture<SendResult<String, Object>>> futures = new
ArrayList<>();
            BatchIdempotentResult batchResult = new BatchIdempotentResult();

            for (OrderEvent order : orders) {
                String messageId = order.getOrderId();

                // Skip already processed messages
                if (processedMessageIds.contains(messageId)) {
                    batchResult.addSkipped(order);
                    continue;
                }

                Message<OrderEvent> message = MessageBuilder
                    .withPayload(order)
                    .setHeader(KafkaHeaders.TOPIC, topic)
                    .setHeader(KafkaHeaders.KEY, messageId)
                    .setHeader("message-id", messageId)
                    .setHeader("batch-id", UUID.randomUUID().toString())
                    .setHeader("batch-timestamp", System.currentTimeMillis())
                    .build();

                CompletableFuture<SendResult<String, Object>> future =
                    idempotentKafkaTemplate.send(message);
                futures.add(future);
            }

            // Combine all futures
            CompletableFuture<Void> allFutures = CompletableFuture.allOf(
                futures.toArray(new CompletableFuture[0]));

            return allFutures.handle((result, ex) -> {
                for (int i = 0; i < futures.size(); i++) {
```

```
                try {
                    SendResult<String, Object> sendResult = futures.get(i).get();
                    OrderEvent order = orders.get(i);
                    batchResult.addSuccess(order, sendResult);
                    processedMessageIds.add(order.getOrderId());
                } catch (Exception e) {
                    batchResult.addFailure(orders.get(i), e);
                }
            }

            logger.info("Batch idempotent send completed: {} successful, {}
failed, {} skipped",
                batchResult.getSuccessCount(),
                batchResult.getFailureCount(),
                batchResult.getSkippedCount());

            return batchResult;
        });
    }

    /**
     * Exactly-once processing simulation
     */
    public void processMessageExactlyOnce(String messageId, Object message,
                                          Runnable processingLogic) {

        logger.info("Processing message exactly once: messageId={}", messageId);

        // Check if already processed
        if (processedMessageIds.contains(messageId)) {
            logger.info("Message already processed exactly once: messageId={}",
messageId);
            return;
        }

        try {
            // Process message
            processingLogic.run();

            // Send confirmation with idempotence
            ProcessingConfirmationEvent confirmation =
ProcessingConfirmationEvent.builder()
                .messageId(messageId)
                .processed(true)
                .processingTimestamp(Instant.now())
                .build();

            sendIdempotentMessage("processing-confirmations", messageId,
confirmation);

            // Mark as processed
            processedMessageIds.add(messageId);

            logger.info("Message processed exactly once: messageId={}",
```

```
messageId);

        } catch (Exception e) {
            logger.error("Exactly-once processing failed: messageId={}",
messageId, e);

            // Send failure confirmation
            ProcessingConfirmationEvent failureConfirmation =
ProcessingConfirmationEvent.builder()
                .messageId(messageId)
                .processed(false)
                .error(e.getMessage())
                .processingTimestamp(Instant.now())
                .build();

            sendIdempotentMessage("processing-confirmations", messageId,
failureConfirmation);
            throw e;
        }
    }

    // Helper methods
    private String generateIdempotenceKey(String messageId, Object message) {
        return messageId + "-" + message.hashCode();
    }

    private void handleSendSuccess(String messageId, SendResult<String, Object>
result) {
        logger.debug("Handling send success: messageId={}", messageId);
    }

    private void handleSendFailure(String messageId, Object message, Throwable ex)
{
        logger.error("Handling send failure: messageId={}", messageId, ex);
    }

    private void handleFinalFailure(String messageId, Object message, Throwable
ex) {
        logger.error("Final failure after all retries: messageId={}", messageId,
ex);

        // Send to dead letter queue or alert
        DeadLetterEvent deadLetter = DeadLetterEvent.builder()
            .messageId(messageId)
            .originalMessage(message)
            .error(ex.getMessage())
            .timestamp(Instant.now())
            .build();

        idempotentKafkaTemplate.send("dead-letter-queue", messageId, deadLetter);
    }
}

// Result classes for batch operations
```

```
@lombok.Data
class BatchIdempotentResult {
    private final List<OrderEvent> successful = new ArrayList<>();
    private final List<OrderEvent> failed = new ArrayList<>();
    private final List<OrderEvent> skipped = new ArrayList<>();
    private final Map<OrderEvent, Exception> failures = new HashMap<>();

    public void addSuccess(OrderEvent order, SendResult<String, Object> result) {
        successful.add(order);
    }

    public void addFailure(OrderEvent order, Exception e) {
        failed.add(order);
        failures.put(order, e);
    }

    public void addSkipped(OrderEvent order) {
        skipped.add(order);
    }

    public int getSuccessCount() { return successful.size(); }
    public int getFailureCount() { return failed.size(); }
    public int getSkippedCount() { return skipped.size(); }
}

@lombok.Data
@lombok.Builder
@lombok.NoArgsConstructor
@lombok.AllArgsConstructor
class ProcessingConfirmationEvent {
    private String messageId;
    private boolean processed;
    private String error;
    private Instant processingTimestamp;
}

@lombok.Data
@lombok.Builder
@lombok.NoArgsConstructor
@lombok.AllArgsConstructor
class DeadLetterEvent {
    private String messageId;
    private Object originalMessage;
    private String error;
    private Instant timestamp;
}
```

Comparisons & Trade-offs

Producer Pattern Comparison

Pattern	Throughput	Latency	Reliability	Complexity	Use Case
Sync Send	Low	High	Highest	Low	Critical operations requiring confirmation
Async Send	High	Low	Medium	Medium	High-throughput, non-critical
Async + Callback	High	Low	High	Medium	High-throughput with error handling
Transactional	Medium	Medium	Highest	High	Cross-topic atomic operations
Idempotent	High	Low	High	Medium	Exactly-once requirements

Serialization Performance Comparison

Serializer	Performance	Size	Schema Evolution	Use Case
String	Fastest	Large	None	Simple text data
JSON	Fast	Medium	Limited	General purpose, human-readable
Avro	Medium	Smallest	Excellent	Schema registry, evolution
Protobuf	Fast	Small	Good	Performance-critical
Custom	Variable	Variable	Full control	Specific requirements

Partitioner Strategy Trade-offs

Strategy	Ordering	Load Balance	Hot Partitions	Use Case
Default Hash	Per-key	Good	Possible	General purpose
Round Robin	None	Excellent	No	Keyless messages
Custom Business	Configurable	Variable	Possible	Business rules
Sticky	None	Good	Temporary	Batching efficiency
Load-Aware	Limited	Excellent	No	Dynamic load balancing

🚨 Common Pitfalls & Best Practices

Common Anti-Patterns

❌ Configuration Mistakes

```
// DON'T - Incorrect transaction configuration
@Bean
public ProducerFactory<String, Object> badTransactionalProducer() {
    Map<String, Object> props = new HashMap<>();
    props.put(ProducerConfig.TRANSACTIONAL_ID_CONFIG, "tx-producer");
```

```
    // MISSING: enable.idempotence=true
    // MISSING: acks=all
    // MISSING: max.in.flight.requests.per.connection=1
    return new DefaultKafkaProducerFactory<>(props);
}

// DON'T - Blocking in async callback
kafkaTemplate.send("topic", message).whenComplete((result, ex) -> {
    if (ex == null) {
        // BLOCKING call in async callback
        String response = restTemplate.getForObject("http://slow-service",
String.class);
        processResponse(response);
    }
});

// DON'T - Ignoring send failures
kafkaTemplate.send("topic", message); // Fire and forget without error handling
```

❌ Transaction Misuse

```
// DON'T - Mixed transaction managers
@Transactional("kafkaTransactionManager")
public void badTransactionMix(OrderEvent order) {
    kafkaTemplate.send("orders", order);

    // This database operation is NOT part of Kafka transaction
    jdbcTemplate.update("INSERT INTO orders...", order.getOrderId());
    // If DB fails, Kafka message still commits!
}

// DON'T - Long-running transactions
@Transactional("kafkaTransactionManager")
public void badLongTransaction() {
    for (int i = 0; i < 10000; i++) { // TOO MANY operations in one transaction
        kafkaTemplate.send("topic", "message-" + i);
        Thread.sleep(100); // Makes transaction even longer
    }
}
```

Production Best Practices

☑ Optimal Configuration Patterns

```
/**
 * Production-ready producer configuration
 */
@Configuration
```

```
public class ProductionProducerConfiguration {

    /**
     * ☑ GOOD - Properly configured transactional producer
     */
    @Bean
    public ProducerFactory<String, Object>
productionTransactionalProducerFactory() {
        Map<String, Object> props = new HashMap<>();

        // Basic configuration
        props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,
"kafka1:9092,kafka2:9092,kafka3:9092");
        props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
StringSerializer.class);
        props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
JsonSerializer.class);

        // Transaction requirements (all must be set together)
        props.put(ProducerConfig.TRANSACTIONAL_ID_CONFIG, "tx-producer");
        props.put(ProducerConfig.ENABLE_IDEMPOTENCE_CONFIG, true);
        props.put(ProducerConfig.ACKS_CONFIG, "all");
        props.put(ProducerConfig.RETRIES_CONFIG, Integer.MAX_VALUE);
        props.put(ProducerConfig.MAX_IN_FLIGHT_REQUESTS_PER_CONNECTION, 1);

        // Performance optimization
        props.put(ProducerConfig.BATCH_SIZE_CONFIG, 16384);
        props.put(ProducerConfig.LINGER_MS_CONFIG, 5);
        props.put(ProducerConfig.COMPRESSION_TYPE_CONFIG, "lz4");

        // Timeout configuration
        props.put(ProducerConfig.REQUEST_TIMEOUT_MS_CONFIG, 30000);
        props.put(ProducerConfig.DELIVERY_TIMEOUT_MS_CONFIG, 120000);
        props.put(ProducerConfig.TRANSACTION_TIMEOUT_CONFIG, 60000);

        DefaultKafkaProducerFactory<String, Object> factory = new
DefaultKafkaProducerFactory<>(props);
        factory.setTransactionIdPrefix("tx-prod-");

        return factory;
    }

    /**
     * ☑ GOOD - Monitoring and metrics
     */
    @Bean
    public ProducerListener<String, Object> productionProducerListener() {
        return new ProducerListener<String, Object>() {

            private final MeterRegistry meterRegistry = Metrics.globalRegistry;
            private final Counter successCounter =
Counter.builder("kafka.producer.success")
                .register(meterRegistry);
            private final Counter errorCounter =
```

```
Counter.builder("kafka.producer.error")
                .register(meterRegistry);
            private final Timer sendTimer =
Timer.builder("kafka.producer.send.time")
                .register(meterRegistry);

            @Override
            public void onSuccess(ProducerRecord<String, Object> record,
RecordMetadata metadata) {
                successCounter.increment(Tags.of("topic", record.topic()));

                // Log only errors and periodic success summaries
                if (successCounter.count() % 1000 == 0) {
                    logger.info("Producer success milestone: {} messages sent",
successCounter.count());
                }
            }

            @Override
            public void onError(ProducerRecord<String, Object> record,
                                RecordMetadata metadata, Exception exception) {
                errorCounter.increment(Tags.of(
                    "topic", record.topic(),
                    "error.type", exception.getClass().getSimpleName()
                ));

                logger.error("Producer error: topic={}, key={}, error={}",
                    record.topic(), record.key(), exception.getMessage());
            }
        };
    }
}

/**
 * ☑ GOOD - Production service patterns
 */
@Service
public class ProductionProducerService {

    @Autowired
    private KafkaTemplate<String, Object> kafkaTemplate;

    /**
     * ☑ GOOD - Async with proper error handling
     */
    public CompletableFuture<SendResult<String, Object>> sendSafely(String topic,
String key, Object message) {
        return kafkaTemplate.send(topic, key, message)
            .orTimeout(10, TimeUnit.SECONDS)
            .whenComplete((result, ex) -> {
                if (ex != null) {
                    handleSendFailure(topic, key, message, ex);
                } else {
                    updateMetrics(topic, result);
```

```
                }
            });
    }

    /**
     * ☑ GOOD - Transaction with proper boundaries
     */
    @Transactional("kafkaTransactionManager")
    public void processOrderTransaction(OrderEvent order) {
        // Keep transaction focused and short
        try {
            // Send related messages atomically
            kafkaTemplate.send("orders", order.getOrderId(), order);
            kafkaTemplate.send("notifications", order.getCustomerId(),
                createNotification(order));
            kafkaTemplate.send("audit", order.getOrderId(),
                createAuditEvent(order));

        } catch (Exception e) {
            logger.error("Transaction failed for order: {}", order.getOrderId(),
e);
            throw e; // Let transaction manager handle rollback
        }
    }

    /**
     * ☑ GOOD - Circuit breaker pattern
     */
    private final CircuitBreaker circuitBreaker =
CircuitBreaker.ofDefaults("kafka-producer");

    public void sendWithCircuitBreaker(String topic, String key, Object message) {
        Supplier<CompletableFuture<SendResult<String, Object>>> sendSupplier =
            CircuitBreaker.decorateSupplier(circuitBreaker, () ->
                kafkaTemplate.send(topic, key, message));

        try {
            CompletableFuture<SendResult<String, Object>> result =
sendSupplier.get();
            result.whenComplete((res, ex) -> {
                if (ex != null) {
                    logger.error("Send failed with circuit breaker: topic={}",
topic, ex);
                }
            });
        } catch (Exception e) {
            logger.error("Circuit breaker open for topic: {}", topic);
            handleCircuitBreakerOpen(topic, key, message);
        }
    }

    /**
     * ☑ GOOD - Batch processing with size limits
     */
```

```
    public void processBatch(List<OrderEvent> orders) {
        final int maxBatchSize = 100;
        final int maxTransactionTime = 30; // seconds

        for (int i = 0; i < orders.size(); i += maxBatchSize) {
            int endIndex = Math.min(i + maxBatchSize, orders.size());
            List<OrderEvent> batch = orders.subList(i, endIndex);

            processBatchWithTimeout(batch, maxTransactionTime);
        }
    }

    @Transactional("kafkaTransactionManager")
    private void processBatchWithTimeout(List<OrderEvent> batch, int
timeoutSeconds) {
        long startTime = System.currentTimeMillis();

        for (OrderEvent order : batch) {
            // Check timeout
            if (System.currentTimeMillis() - startTime > timeoutSeconds * 1000) {
                logger.warn("Batch processing timeout, stopping at order: {}",
order.getOrderId());
                break;
            }

            kafkaTemplate.send("orders", order.getOrderId(), order);
        }
    }

    // Helper methods with proper error handling
    private void handleSendFailure(String topic, String key, Object message,
Throwable ex) {
        // Log error with context
        logger.error("Send failure: topic={}, key={}, error={}", topic, key,
ex.getMessage());

        // Implement appropriate failure handling
        if (isCriticalTopic(topic)) {
            alertOperationsTeam(topic, key, ex);
        }

        if (isRetriable(ex)) {
            scheduleRetry(topic, key, message);
        } else {
            sendToDeadLetterQueue(topic, key, message, ex);
        }
    }

    private boolean isCriticalTopic(String topic) {
        return topic.contains("orders") || topic.contains("payments");
    }

    private boolean isRetriable(Throwable ex) {
        return ex instanceof TimeoutException ||
```

```
                ex instanceof org.apache.kafka.common.errors.RetriableException;
    }

    private void alertOperationsTeam(String topic, String key, Throwable ex) {
        // Implement alerting (Slack, PagerDuty, email, etc.)
        logger.error("🚨 CRITICAL: Producer failure on topic: {}, key: {}",
topic, key, ex);
    }

    private void scheduleRetry(String topic, String key, Object message) {
        // Implement retry with exponential backoff
        logger.info("Scheduling retry for: topic={}, key={}", topic, key);
    }

    private void sendToDeadLetterQueue(String topic, String key, Object message,
Throwable ex) {
        // Send to DLQ
        String dlqTopic = topic + ".DLT";
        kafkaTemplate.send(dlqTopic, key, message);
    }

    private void updateMetrics(String topic, SendResult<String, Object> result) {
        // Update success metrics
    }

    private void handleCircuitBreakerOpen(String topic, String key, Object
message) {
        // Handle circuit breaker open state
        logger.warn("Circuit breaker open, using fallback for: topic={}", topic);
    }
}
```

☑ Resource Management Best Practices

```
/**
 * ☑ GOOD - Proper resource management and lifecycle
 */
@Configuration
public class ProducerLifecycleConfiguration {

    /**
     * Graceful shutdown configuration
     */
    @Bean
    public ProducerFactory<String, Object> gracefulShutdownProducerFactory() {
        DefaultKafkaProducerFactory<String, Object> factory =
            new DefaultKafkaProducerFactory<>(producerConfigs());

        // Enable graceful shutdown
        factory.setCloseTimeout(Duration.ofSeconds(30));
```

```
        return factory;
    }

    /**
     * Application shutdown handler
     */
    @EventListener
    public void handleContextClosed(ContextClosedEvent event) {
        logger.info("Application shutdown - flushing Kafka producers");

        // Flush any pending messages
        kafkaTemplate.flush();

        // Wait for in-flight messages
        try {
            Thread.sleep(5000);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }

    /**
     * Health check for producer
     */
    @Component
    public static class ProducerHealthIndicator implements HealthIndicator {

        @Autowired
        private KafkaTemplate<String, Object> kafkaTemplate;

        @Override
        public Health health() {
            try {
                // Test producer connectivity
                kafkaTemplate.partitionsFor("__health_check_topic__");

                return Health.up()
                    .withDetail("producer", "Available")
                    .build();
            } catch (Exception e) {
                return Health.down()
                    .withDetail("producer", "Unavailable")
                    .withDetail("error", e.getMessage())
                    .build();
            }
        }
    }
}
```

Real-World Use Cases

E-commerce Order Processing Pipeline

```
/**
 * Production e-commerce order processing with Spring Kafka producers
 */
@Service
@Transactional("kafkaTransactionManager")
public class EcommerceOrderProcessingService {

    @Autowired
    private KafkaTemplate<String, Object> transactionalKafkaTemplate;

    @Autowired
    private KafkaTemplate<String, Object> idempotentKafkaTemplate;

    /**
     * Process complete order lifecycle with transactions
     */
    public void processOrderComplete(OrderEvent order) {
        logger.info("Processing complete order: {}", order.getOrderId());

        try {
            // All messages sent atomically
            transactionalKafkaTemplate.send("order-created", order.getOrderId(),
order);

            // Inventory reservation
            InventoryReservationEvent reservation =
InventoryReservationEvent.builder()
                .orderId(order.getOrderId())
                .productId(order.getProductId())
                .quantity(order.getQuantity())
                .reservationTime(Instant.now())
                .build();
            transactionalKafkaTemplate.send("inventory-reservations",
order.getOrderId(), reservation);

            // Payment processing
            PaymentRequestEvent payment = PaymentRequestEvent.builder()
                .orderId(order.getOrderId())
                .customerId(order.getCustomerId())
                .amount(order.getAmount())
                .paymentMethod(order.getPaymentMethod())
                .build();
            transactionalKafkaTemplate.send("payment-requests",
order.getOrderId(), payment);

            // Customer notification
            NotificationEvent notification = NotificationEvent.builder()
                .customerId(order.getCustomerId())
                .type("ORDER_PLACED")
                .message("Your order " + order.getOrderId() + " has been placed
```

```
successfully")
                .channel("EMAIL")
                .build();
            transactionalKafkaTemplate.send("customer-notifications",
order.getCustomerId(), notification);

            logger.info("Order processing completed successfully: {}",
order.getOrderId());

        } catch (Exception e) {
            logger.error("Order processing failed: {}", order.getOrderId(), e);
            throw e; // Transaction will be rolled back
        }
    }
}
```

Financial Transaction Processing

```
/**
 * Financial transaction processing with exactly-once semantics
 */
@Service
public class FinancialTransactionService {

    @Autowired
    private KafkaTemplate<String, Object> transactionalKafkaTemplate;

    /**
     * Process financial transfer with ACID guarantees
     */
    @Transactional("kafkaTransactionManager")
    public void processMoneyTransfer(MoneyTransferRequest transfer) {
        logger.info("Processing money transfer: {}", transfer.getTransactionId());

        try {
            // Debit from source account
            AccountDebitEvent debit = AccountDebitEvent.builder()
                .transactionId(transfer.getTransactionId())
                .accountId(transfer.getFromAccount())
                .amount(transfer.getAmount())
                .currency(transfer.getCurrency())
                .timestamp(Instant.now())
                .build();
            transactionalKafkaTemplate.send("account-debits",
transfer.getFromAccount(), debit);

            // Credit to destination account
            AccountCreditEvent credit = AccountCreditEvent.builder()
                .transactionId(transfer.getTransactionId())
                .accountId(transfer.getToAccount())
                .amount(transfer.getAmount())
```

```
                .currency(transfer.getCurrency())
                .timestamp(Instant.now())
                .build();
            transactionalKafkaTemplate.send("account-credits",
transfer.getToAccount(), credit);

            // Transaction log
            TransactionLogEvent log = TransactionLogEvent.builder()
                .transactionId(transfer.getTransactionId())
                .type("MONEY_TRANSFER")
                .fromAccount(transfer.getFromAccount())
                .toAccount(transfer.getToAccount())
                .amount(transfer.getAmount())
                .status("COMPLETED")
                .timestamp(Instant.now())
                .build();
            transactionalKafkaTemplate.send("transaction-logs",
transfer.getTransactionId(), log);

            logger.info("Money transfer completed: {}",
transfer.getTransactionId());

        } catch (Exception e) {
            logger.error("Money transfer failed: {}", transfer.getTransactionId(),
e);
            throw e;
        }
    }
}
```

IoT Data Streaming Pipeline

```
/**
 * IoT sensor data processing with custom partitioning
 */
@Service
public class IoTDataStreamingService {

    @Autowired
    private KafkaTemplate<String, Object> kafkaTemplate;

    /**
     * Process sensor data with geographic partitioning
     */
    public void processSensorData(List<SensorReading> readings) {
        logger.info("Processing {} sensor readings", readings.size());

        // Group by sensor type for efficient processing
        Map<String, List<SensorReading>> bySensorType = readings.stream()
            .collect(Collectors.groupingBy(SensorReading::getSensorType));
```

```
        bySensorType.forEach((sensorType, typeReadings) -> {
            processSensorTypeData(sensorType, typeReadings);
        });
    }

    private void processSensorTypeData(String sensorType, List<SensorReading>
readings) {
        String topic = "sensor-data-" + sensorType.toLowerCase();

        List<CompletableFuture<SendResult<String, Object>>> futures = new
ArrayList<>();

        for (SensorReading reading : readings) {
            // Use sensor ID as key for partition affinity
            String key = reading.getSensorId();

            CompletableFuture<SendResult<String, Object>> future =
                kafkaTemplate.send(topic, key, reading);

            futures.add(future);
        }

        // Handle batch completion
        CompletableFuture.allOf(futures.toArray(new CompletableFuture[0]))
            .whenComplete((result, ex) -> {
                if (ex != null) {
                    logger.error("Batch sensor data processing failed for type:
{}", sensorType, ex);
                } else {
                    logger.info("Successfully processed {} {} sensor readings",
                        readings.size(), sensorType);
                }
            });
    }
}

// Domain objects for use cases
@lombok.Data
@lombok.Builder
@lombok.NoArgsConstructor
@lombok.AllArgsConstructor
class InventoryReservationEvent {
    private String orderId;
    private String productId;
    private int quantity;
    private Instant reservationTime;
}

@lombok.Data
@lombok.Builder
@lombok.NoArgsConstructor
@lombok.AllArgsConstructor
class PaymentRequestEvent {
    private String orderId;
```

```
    private String customerId;
    private java.math.BigDecimal amount;
    private String paymentMethod;
}

@lombok.Data
@lombok.Builder
@lombok.NoArgsConstructor
@lombok.AllArgsConstructor
class MoneyTransferRequest {
    private String transactionId;
    private String fromAccount;
    private String toAccount;
    private java.math.BigDecimal amount;
    private String currency;
}

@lombok.Data
@lombok.Builder
@lombok.NoArgsConstructor
@lombok.AllArgsConstructor
class AccountDebitEvent {
    private String transactionId;
    private String accountId;
    private java.math.BigDecimal amount;
    private String currency;
    private Instant timestamp;
}

@lombok.Data
@lombok.Builder
@lombok.NoArgsConstructor
@lombok.AllArgsConstructor
class AccountCreditEvent {
    private String transactionId;
    private String accountId;
    private java.math.BigDecimal amount;
    private String currency;
    private Instant timestamp;
}

@lombok.Data
@lombok.Builder
@lombok.NoArgsConstructor
@lombok.AllArgsConstructor
class TransactionLogEvent {
    private String transactionId;
    private String type;
    private String fromAccount;
    private String toAccount;
    private java.math.BigDecimal amount;
    private String status;
    private Instant timestamp;
}
```

```
@lombok.Data
@lombok.Builder
@lombok.NoArgsConstructor
@lombok.AllArgsConstructor
class SensorReading {
    private String sensorId;
    private String sensorType;
    private double value;
    private String unit;
    private double latitude;
    private double longitude;
    private Instant timestamp;
}
```

Version Highlights

Spring Kafka Producer Evolution Timeline

Version	Release	Key Producer Features
3.1.x	2024	Enhanced transaction performance, improved error handling
3.0.x	2023	Spring Boot 3 support, native compilation improvements
2.9.x	2022	Better batch processing, enhanced callback mechanisms
2.8.x	2022	Improved transactional templates, ReplyingKafkaTemplate enhancements
2.7.x	2021	Enhanced retry mechanisms, better error recovery
2.6.x	2021	Non-blocking retries, improved DLT handling
2.5.x	2020	Enhanced security features, OAuth integration
2.4.x	2020	Improved batch processing, performance optimizations
2.3.x	2019	Request-reply patterns, ReplyingKafkaTemplate introduction
2.2.x	2018	Transaction support, KafkaTemplate improvements

Modern Producer Features (2023-2025)

Spring Kafka 3.1+ Producer Enhancements:

- **Improved Transaction Performance**: Better throughput in transactional scenarios
- **Enhanced Error Handling**: More sophisticated retry and recovery mechanisms
- **Better Observability**: Improved metrics and tracing integration
- **Native Compilation**: GraalVM support for faster startup and lower memory usage

Key Configuration Evolution:

- **Simplified Transaction Setup**: Reduced boilerplate for transactional producers

- **Better Default Configurations**: Production-ready defaults for most scenarios
- **Enhanced Serialization**: Improved JSON and Avro integration
- **Performance Tuning**: Better batch processing and memory management

---

🔗 Additional Resources

📚 Official Documentation

- Spring Kafka Producer Reference
- Apache Kafka Producer Configuration
- Spring Kafka Transactions

🎓 Learning Resources

- Confluent Spring Kafka Course
- Spring Kafka Examples
- Kafka Producer Best Practices

🛠️ Development Tools

- Kafka Tool - GUI for Kafka cluster management
- Conduktor - Modern Kafka desktop client
- Schema Registry UI - Web interface for Schema Registry

Last Updated: September 2025
Spring Kafka Version Coverage: 3.1.x
Spring Boot Compatibility: 3.2.x
Apache Kafka Version: 3.6.x

> 💡 **Pro Tip**: Start with idempotent producers for reliability, use transactions only when you need atomicity across multiple topics, and always implement proper error handling with callbacks. The combination of Spring's abstractions with Kafka's powerful producer features provides an excellent foundation for building robust, high-performance message-driven applications.