Spring Kafka Consumer Side: Complete Developer Guide

A comprehensive guide covering Spring Kafka consumer implementation, from @KafkaListener basics to advanced container configurations with extensive Java examples and best practices.

Table of Contents

- 📨 @KafkaListener Annotation
  - Single-topic vs Multi-topic Listeners
  - Concurrency Handling
  - Message Filtering Strategies
- 🏗 Listener Containers
  - ConcurrentMessageListenerContainer
  - BatchMessageListenerContainer
  - Manual vs Auto Ack Modes
- ⚙ Consumer Configuration
  - Deserializers
  - Rebalance Listeners
  - Error Handling Strategies
- 📊 Comparisons & Trade-offs
- 🚨 Common Pitfalls & Best Practices
- 🌍 Real-World Use Cases
- 📈 Version Highlights

📨 @KafkaListener Annotation

What is @KafkaListener?

Simple Explanation: @KafkaListener is a declarative annotation that turns any Spring bean method into a Kafka message consumer, handling the complexities of consumer lifecycle, partition assignment, and message deserialization automatically.

Problem It Solves:

- **Boilerplate Elimination**: Removes the need for manual consumer creation and management
- **Declarative Configuration**: Configures consumers through annotations rather than imperative code
- **Spring Integration**: Seamless integration with Spring's dependency injection and transaction management
- **Error Handling**: Built-in error handling and retry mechanisms
- **Concurrency Management**: Automatic scaling based on partition count and configuration

Internal Architecture:

```
@KafkaListener Architecture:

┌─────────────────────────────────────────────────────────────────┐
│                @KafkaListener Method                            │
├─────────────────────────────────────────────────────────────────┤
│  ┌─────────────────┐  ┌─────────────────┐  ┌─────────────────┐  │
│  │    Method       │  │ Parameter       │  │ Return Value    │  │
│  │ Annotations     │  │ Injection       │  │ Handling        │  │
│  └─────────────────┘  └─────────────────┘  └─────────────────┘  │
├─────────────────────────────────────────────────────────────────┤
│              KafkaListenerEndpointRegistry                      │
│  ┌─────────────────┐  ┌─────────────────┐  ┌─────────────────┐  │
│  │    Endpoint     │  │   Container     │  │   Lifecycle     │  │
│  │ Registration    │  │   Factory       │  │   Management    │  │
│  └─────────────────┘  └─────────────────┘  └─────────────────┘  │
├─────────────────────────────────────────────────────────────────┤
│                MessageListenerContainer                         │
│  ┌─────────────────┐  ┌─────────────────┐  ┌─────────────────┐  │
│  │   Consumer      │  │   Message       │  │    Error        │  │
│  │   Management    │  │   Processing    │  │   Handling      │  │
│  └─────────────────┘  └─────────────────┘  └─────────────────┘  │
└─────────────────────────────────────────────────────────────────┘
                                ↓
┌─────────────────────────────────────────────────────────────────┐
│                 Native Kafka Consumer                           │
└─────────────────────────────────────────────────────────────────┘
```

Single-topic vs Multi-topic Listeners

Comprehensive Single-Topic Listener Patterns

```java
import org.springframework.kafka.annotation.KafkaListener;
import org.springframework.kafka.annotation.PartitionOffset;
import org.springframework.kafka.annotation.TopicPartition;
import org.springframework.kafka.support.Acknowledgment;
import org.springframework.kafka.support.KafkaHeaders;
import org.springframework.messaging.handler.annotation.Header;
import org.springframework.messaging.handler.annotation.Payload;
import org.springframework.stereotype.Component;

import org.apache.kafka.clients.consumer.ConsumerRecord;
import org.apache.kafka.clients.consumer.Consumer;

import java.util.List;
import java.time.Instant;

/**
 * Comprehensive single-topic listener patterns
 */
@Component
```

```
@lombok.extern.slf4j.Slf4j
public class SingleTopicKafkaListeners {

    /**
     * Basic single-topic listener
     * Simplest form - listens to all partitions of a topic
     */
    @KafkaListener(
        topics = "orders",
        groupId = "order-processing-group"
    )
    public void handleOrderEvents(OrderEvent orderEvent) {
        log.info("Processing order: orderId={}, customerId={}, amount={}",
            orderEvent.getOrderId(),
            orderEvent.getCustomerId(),
            orderEvent.getAmount());

        try {
            processOrder(orderEvent);
            log.info("Successfully processed order: {}", orderEvent.getOrderId());
        } catch (Exception e) {
            log.error("Failed to process order: {}", orderEvent.getOrderId(), e);
            throw e; // Let error handler manage retry/recovery
        }
    }

    /**
     * Single-topic listener with complete message context
     * Access to all Kafka metadata for detailed processing
     */
    @KafkaListener(
        topics = "notifications",
        groupId = "notification-service",
        containerFactory = "kafkaListenerContainerFactory"
    )
    public void handleNotificationWithContext(
            @Payload NotificationEvent notification,
            @Header(KafkaHeaders.RECEIVED_TOPIC) String topic,
            @Header(KafkaHeaders.RECEIVED_PARTITION_ID) int partition,
            @Header(KafkaHeaders.OFFSET) long offset,
            @Header(KafkaHeaders.RECEIVED_TIMESTAMP) long timestamp,
            @Header(KafkaHeaders.RECEIVED_KEY) String key,
            @Header(value = "correlation-id", required = false) String
correlationId,
            ConsumerRecord<String, NotificationEvent> record,
            Acknowledgment acknowledgment) {

        log.info("Processing notification: topic={}, partition={}, offset={},
timestamp={}, correlationId={}",
            topic, partition, offset, timestamp, correlationId);

        try {
            // Process with full context
            processNotificationWithMetadata(notification, record);
            // Manual acknowledgment after successful processing
            acknowledgment.acknowledge();

            log.info("Successfully processed notification: userId={}, type={}",
                notification.getUserId(), notification.getType());

        } catch (Exception e) {
            log.error("Failed to process notification: partition={}, offset={}",
                partition, offset, e);
            // Don't acknowledge - will be retried based on error handler
configuration
            throw e;
        }
    }

    /**
     * Single-topic listener for specific partitions
     * Manual partition assignment for ordered processing
     */
    @KafkaListener(
        topicPartitions = {
            @TopicPartition(topic = "user-events", partitions = {"0", "1", "2"}),
            @TopicPartition(topic = "user-events",
                partitions = "3",
                partitionOffsets = @PartitionOffset(partition = "3", initialOffset
= "100"))
        },
        groupId = "user-service-specific-partitions",
        containerFactory = "manualPartitionContainerFactory"
    )
    public void handleUserEventsFromSpecificPartitions(
            @Payload UserEvent userEvent,
            @Header(KafkaHeaders.RECEIVED_PARTITION_ID) int partition,
            @Header(KafkaHeaders.OFFSET) long offset) {

        log.info("Processing user event from partition {}: userId={}, action={},
offset={}",
            partition, userEvent.getUserId(), userEvent.getAction(), offset);

        try {
            // Partition-specific processing logic
            processUserEventByPartition(userEvent, partition);

            // Update processing metrics per partition
            updatePartitionMetrics(partition, offset);

        } catch (Exception e) {
            log.error("Failed to process user event from partition {}: userId={}",
                partition, userEvent.getUserId(), e);
            throw e;
        }
    }
```

```java
    /**
     * Single-topic listener with message filtering
     * Process only specific message types
     */
    @KafkaListener(
        topics = "events",
        groupId = "filtered-events-processor",
        containerFactory = "filteredKafkaListenerContainerFactory",
        filter = "eventFilter"
    )
    public void handleFilteredEvents(@Payload EventMessage event) {
        log.info("Processing filtered event: type={}, id={}",
            event.getEventType(), event.getEventId());

        // Only specific event types reach here based on filter
        processFilteredEvent(event);
    }

    /**
     * Single-topic listener with custom properties
     * Override consumer properties for specific listeners
     */
    @KafkaListener(
        topics = "high-volume-events",
        groupId = "high-volume-processor",
        properties = {
            "max.poll.records=1000",
            "fetch.min.bytes=1048576", // 1MB
            "fetch.max.wait.ms=500",
            "session.timeout.ms=30000",
            "heartbeat.interval.ms=10000"
        }
    )
    public void handleHighVolumeEvents(@Payload List<EventMessage> events) {
        log.info("Processing high-volume batch: {} events", events.size());

        // Batch processing for high throughput
        processBatchEvents(events);
    }

    /**
     * Single-topic listener with error recovery
     * Custom error handling for specific use cases
     */
    @KafkaListener(
        topics = "critical-orders",
        groupId = "critical-order-processor",
        containerFactory = "errorHandlingContainerFactory"
    )
    public void handleCriticalOrders(@Payload OrderEvent order,
                                       ConsumerRecord<String, OrderEvent> record) {
        log.info("Processing critical order: orderId={}, amount={}",
            order.getOrderId(), order.getAmount());
        try {
            validateCriticalOrder(order);
            processCriticalOrder(order);

        } catch (ValidationException e) {
            log.error("Critical order validation failed: orderId={}",
order.getOrderId(), e);
            sendToValidationFailureTopic(order, e);
            throw e;
        } catch (Exception e) {
            log.error("Critical order processing failed: orderId={}",
order.getOrderId(), e);
            alertOperationsTeam(order, e);
            throw e;
        }
    }

    /**
     * Single-topic listener with conditional processing
     * Different handling based on message content
     */
    @KafkaListener(
        topics = "payment-events",
        groupId = "payment-processor"
    )
    public void handlePaymentEvents(@Payload PaymentEvent payment) {
        log.info("Processing payment: paymentId={}, amount={}, method={}",
            payment.getPaymentId(), payment.getAmount(), payment.getMethod());

        // Route processing based on payment method
        switch (payment.getMethod().toUpperCase()) {
            case "CREDIT_CARD" -> processCreditCardPayment(payment);
            case "BANK_TRANSFER" -> processBankTransferPayment(payment);
            case "DIGITAL_WALLET" -> processDigitalWalletPayment(payment);
            default -> {
                log.warn("Unknown payment method: {}", payment.getMethod());
                processGenericPayment(payment);
            }
        }
    }

    // Business logic methods
    private void processOrder(OrderEvent order) {
        log.debug("Processing order business logic for: {}", order.getOrderId());
    }

    private void processNotificationWithMetadata(NotificationEvent notification,
                                                ConsumerRecord<String,
NotificationEvent> record) {
        log.debug("Processing notification with metadata: userId={}, timestamp=
{}",
            notification.getUserId(), record.timestamp());
    }
    private void processUserEventByPartition(UserEvent event, int partition) {
        log.debug("Partition-specific processing for user {} on partition {}",
            event.getUserId(), partition);
    }

    private void updatePartitionMetrics(int partition, long offset) {
        log.debug("Updating metrics for partition {} at offset {}", partition,
offset);
    }

    private void processFilteredEvent(EventMessage event) {
        log.debug("Processing filtered event: {}", event.getEventId());
    }

    private void processBatchEvents(List<EventMessage> events) {
        log.debug("Processing batch of {} events", events.size());
    }

    private void validateCriticalOrder(OrderEvent order) throws
ValidationException {
        if (order.getAmount().compareTo(java.math.BigDecimal.ZERO) <= 0) {
            throw new ValidationException("Invalid order amount");
        }
    }

    private void processCriticalOrder(OrderEvent order) {
        log.debug("Processing critical order: {}", order.getOrderId());
    }

    private void sendToValidationFailureTopic(OrderEvent order, Exception e) {
        log.info("Sending order to validation failure topic: {}",
order.getOrderId());
    }

    private void alertOperationsTeam(OrderEvent order, Exception e) {
        log.error("🚨 CRITICAL ORDER FAILURE - alerting operations: {}",
order.getOrderId());
    }

    private void processCreditCardPayment(PaymentEvent payment) {
        log.debug("Processing credit card payment: {}", payment.getPaymentId());
    }

    private void processBankTransferPayment(PaymentEvent payment) {
        log.debug("Processing bank transfer payment: {}", payment.getPaymentId());
    }

    private void processDigitalWalletPayment(PaymentEvent payment) {
        log.debug("Processing digital wallet payment: {}",
payment.getPaymentId());
    }

    private void processGenericPayment(PaymentEvent payment) {
        log.debug("Processing generic payment: {}", payment.getPaymentId());
    }
}

/**
 * Custom validation exception
 */
class ValidationException extends Exception {
    public ValidationException(String message) {
        super(message);
    }
}
```

Multi-Topic Listener Patterns

```
import org.springframework.kafka.annotation.KafkaListener;
import org.springframework.kafka.support.KafkaHeaders;
import org.springframework.messaging.handler.annotation.Header;
import org.springframework.messaging.handler.annotation.Payload;

/**
 * Comprehensive multi-topic listener patterns
 */
@Component
@lombok.extern.slf4j.Slf4j
public class MultiTopicKafkaListeners {

    /**
     * Basic multi-topic listener
     * Single listener handling multiple topics
     */
    @KafkaListener(
        topics = {"orders", "payments", "notifications"},
        groupId = "multi-topic-processor"
    )
    public void handleMultipleTopics(
            @Payload Object message,
            @Header(KafkaHeaders.RECEIVED_TOPIC) String topic) {

        log.info("Processing message from topic: {}", topic);

        // Route based on topic
        switch (topic) {
            case "orders" -> handleOrderMessage((OrderEvent) message);
            case "payments" -> handlePaymentMessage((PaymentEvent) message);
            case "notifications" -> handleNotificationMessage((NotificationEvent)
message);
            default -> log.warn("Unknown topic: {}", topic);
        }
    }

    /**
```

```
     * Multi-topic listener with pattern matching
     * Use regex patterns to match topic names
     */
    @KafkaListener(
        topicPattern = "user-events-.*",
        groupId = "user-events-aggregator"
    )
    public void handleUserEventTopics(
            @Payload UserEvent userEvent,
            @Header(KafkaHeaders.RECEIVED_TOPIC) String topic,
            @Header(KafkaHeaders.RECEIVED_PARTITION_ID) int partition) {

        log.info("Processing user event from topic: {}, partition: {}, userId:
{}",
            topic, partition, userEvent.getUserId());

        // Extract region from topic name (e.g., user-events-us-west)
        String region = extractRegionFromTopic(topic);
        processUserEventByRegion(userEvent, region);
    }

    /**
     * Multi-topic listener with type-safe message handling
     * Handle different message types from multiple topics
     */
    @KafkaListener(
        topics = {"order-events", "payment-events", "shipping-events"},
        groupId = "event-aggregator",
        containerFactory = "multiTypeContainerFactory"
    )
    public void handleBusinessEvents(
            @Payload Object event,
            @Header(KafkaHeaders.RECEIVED_TOPIC) String topic,
            @Header(value = "event-type", required = false) String eventType,
            ConsumerRecord<String, Object> record) {

        log.info("Processing business event: topic={}, eventType={}", topic,
eventType);

        try {
            // Type-safe message processing
            if (event instanceof OrderEvent orderEvent) {
                processBusinessOrderEvent(orderEvent, topic);
            } else if (event instanceof PaymentEvent paymentEvent) {
                processBusinessPaymentEvent(paymentEvent, topic);
            } else if (event instanceof ShippingEvent shippingEvent) {
                processBusinessShippingEvent(shippingEvent, topic);
            } else {
                log.warn("Unknown event type from topic {}: {}", topic,
event.getClass());
                handleUnknownBusinessEvent(event, topic, record);
            }

        } catch (Exception e) {
            log.error("Failed to process business event from topic: {}", topic,
e);
            throw e;
        }
    }

    /**
     * Multi-topic listener with concurrency optimization
     * Different concurrency for different topic patterns
     */
    @KafkaListener(
        topics = {"high-throughput-events", "batch-processing-events"},
        groupId = "high-throughput-processor",
        concurrency = "8",
        containerFactory = "highThroughputContainerFactory"
    )
    public void handleHighThroughputMultiTopics(
            @Payload List<Object> messages,
            @Header(KafkaHeaders.RECEIVED_TOPIC) List<String> topics,
            @Header(KafkaHeaders.RECEIVED_PARTITION_ID) List<Integer> partitions)
{

        log.info("Processing high-throughput batch: {} messages across {} unique
topics",
            messages.size(), topics.stream().distinct().count());

        // Group messages by topic for efficient processing
        Map<String, List<Object>> messagesByTopic = new HashMap<>();
        for (int i = 0; i < messages.size(); i++) {
            String topic = topics.get(i);
            messagesByTopic.computeIfAbsent(topic, k -> new ArrayList<>
()).add(messages.get(i));
        }

        // Process each topic's messages
        messagesByTopic.forEach(this::processBatchByTopic);
    }

    /**
     * Multi-topic listener with advanced routing
     * Complex routing logic based on topic and message content
     */
    @KafkaListener(
        topicPattern = "(orders|payments|refunds)-.*",
        groupId = "financial-processor",
        containerFactory = "financialProcessingContainerFactory"
    )
    public void handleFinancialTopics(
            @Payload Object message,
            @Header(KafkaHeaders.RECEIVED_TOPIC) String topic,
            @Header(KafkaHeaders.RECEIVED_KEY) String key,
            @Header(value = "business-unit", required = false) String
businessUnit,
            @Header(value = "priority", required = false) String priority) {
```

```
        log.info("Processing financial message: topic={}, key={}, businessUnit={}, 
priority={}",
            topic, key, businessUnit, priority);

        try {
            // Route based on topic type and metadata
            FinancialMessageType messageType = 
determineFinancialMessageType(topic);
            ProcessingPriority processingPriority = 
determineProcessingPriority(priority);

            switch (messageType) {
                case ORDER -> processFinancialOrder((OrderEvent) message, 
processingPriority);
                case PAYMENT -> processFinancialPayment((PaymentEvent) message, 
processingPriority);
                case REFUND -> processFinancialRefund((RefundEvent) message, 
processingPriority);
                default -> log.warn("Unknown financial message type from topic: 
{}", topic);
            }

        } catch (Exception e) {
            log.error("Failed to process financial message from topic: {}", topic, 
e);
            handleFinancialProcessingError(message, topic, e);
            throw e;
        }
    }

    /**
     * Multi-topic listener with dead letter topic handling
     * Centralized error handling for multiple topics
     */
    @KafkaListener(
        topics = {"orders.DLT", "payments.DLT", "notifications.DLT"},
        groupId = "dead-letter-processor"
    )
    public void handleDeadLetterTopics(
            @Payload Object failedMessage,
            @Header(KafkaHeaders.RECEIVED_TOPIC) String dltTopic,
            @Header(value = "kafka_dlt-original-topic", required = false) String 
originalTopic,
            @Header(value = "kafka_dlt-exception-message", required = false) 
String errorMessage,
            @Header(value = "kafka_dlt-exception-stacktrace", required = false) 
String stackTrace) {

        log.warn("Processing dead letter message: originalTopic={}, dltTopic={}, 
error={}",
            originalTopic, dltTopic, errorMessage);

        try {
            // Attempt to recover or route to appropriate error handling
            DeadLetterProcessingResult result = processDeadLetterMessage(
                failedMessage, originalTopic, errorMessage);

            if (result.isRecovered()) {
                log.info("Successfully recovered dead letter message from: {}",
originalTopic);
                republishRecoveredMessage(failedMessage, originalTopic);
            } else {
                log.error("Failed to recover dead letter message from: {}",
originalTopic);
                archiveUnrecoverableMessage(failedMessage, originalTopic,
errorMessage);
            }

        } catch (Exception e) {
            log.error("Error processing dead letter message from: {}",
originalTopic, e);
            // Final fallback - could alert operations team
            alertDeadLetterProcessingFailure(originalTopic, e);
        }
    }

    // Helper methods
    private void handleOrderMessage(OrderEvent order) {
        log.debug("Handling order message: {}", order.getOrderId());
    }

    private void handlePaymentMessage(PaymentEvent payment) {
        log.debug("Handling payment message: {}", payment.getPaymentId());
    }

    private void handleNotificationMessage(NotificationEvent notification) {
        log.debug("Handling notification message: {}", notification.getUserId());
    }

    private String extractRegionFromTopic(String topic) {
        // Extract region from topic name like "user-events-us-west"
        String[] parts = topic.split("-");
        return parts.length >= 3 ? parts[2] + "-" + parts[3] : "unknown";
    }

    private void processUserEventByRegion(UserEvent event, String region) {
        log.debug("Processing user event for region {}: {}", region,
event.getUserId());
    }

    private void processBusinessOrderEvent(OrderEvent order, String topic) {
        log.debug("Processing business order event from {}: {}", topic,
order.getOrderId());
    }

    private void processBusinessPaymentEvent(PaymentEvent payment, String topic) {
        log.debug("Processing business payment event from {}: {}", topic,
payment.getPaymentId());
    }

    private void processBusinessShippingEvent(ShippingEvent shipping, String
topic) {
        log.debug("Processing business shipping event from {}: {}", topic,
shipping.getShipmentId());
    }

    private void handleUnknownBusinessEvent(Object event, String topic,
ConsumerRecord<String, Object> record) {
        log.warn("Handling unknown business event from {}: {}", topic,
event.getClass());
    }

    private void processBatchByTopic(String topic, List<Object> messages) {
        log.debug("Processing batch of {} messages from topic: {}",
messages.size(), topic);
    }

    private FinancialMessageType determineFinancialMessageType(String topic) {
        if (topic.startsWith("orders")) return FinancialMessageType.ORDER;
        if (topic.startsWith("payments")) return FinancialMessageType.PAYMENT;
        if (topic.startsWith("refunds")) return FinancialMessageType.REFUND;
        return FinancialMessageType.UNKNOWN;
    }

    private ProcessingPriority determineProcessingPriority(String priority) {
        return switch (priority) {
            case "HIGH" -> ProcessingPriority.HIGH;
            case "MEDIUM" -> ProcessingPriority.MEDIUM;
            case "LOW" -> ProcessingPriority.LOW;
            default -> ProcessingPriority.NORMAL;
        };
    }

    private void processFinancialOrder(OrderEvent order, ProcessingPriority
priority) {
        log.debug("Processing financial order with priority {}: {}", priority,
order.getOrderId());
    }

    private void processFinancialPayment(PaymentEvent payment, ProcessingPriority
priority) {
        log.debug("Processing financial payment with priority {}: {}", priority,
payment.getPaymentId());
    }

    private void processFinancialRefund(RefundEvent refund, ProcessingPriority
priority) {
        log.debug("Processing financial refund with priority {}: {}", priority,
refund.getRefundId());
    }
```

```java
    private void handleFinancialProcessingError(Object message, String topic, Exception e) {
        log.error("Handling financial processing error for topic: {}", topic);
    }

    private DeadLetterProcessingResult processDeadLetterMessage(Object message, String originalTopic, String error) {
        // Implement dead letter recovery logic
        return new DeadLetterProcessingResult(false);
    }

    private void republishRecoveredMessage(Object message, String originalTopic) {
        log.info("Republishing recovered message to: {}", originalTopic);
    }

    private void archiveUnrecoverableMessage(Object message, String originalTopic, String error) {
        log.warn("Archiving unrecoverable message from: {}", originalTopic);
    }

    private void alertDeadLetterProcessingFailure(String originalTopic, Exception e) {
        log.error("🚨 DEAD LETTER PROCESSING FAILURE: {}", originalTopic);
    }
}

// Enums and helper classes
enum FinancialMessageType {
    ORDER, PAYMENT, REFUND, UNKNOWN
}

enum ProcessingPriority {
    HIGH, MEDIUM, NORMAL, LOW
}

@lombok.Data
@lombok.AllArgsConstructor
class DeadLetterProcessingResult {
    private boolean recovered;
}

// Additional domain objects
@lombok.Data
@lombok.Builder
@lombok.NoArgsConstructor
@lombok.AllArgsConstructor
class ShippingEvent {
    private String shipmentId;
    private String orderId;
    private String carrier;
    private String trackingNumber;
    private Instant timestamp;
}
```

```java
@lombok.Data
@lombok.Builder
@lombok.NoArgsConstructor
@lombok.AllArgsConstructor
class RefundEvent {
    private String refundId;
    private String paymentId;
    private String orderId;
    private java.math.BigDecimal amount;
    private String reason;
    private Instant timestamp;
}

@lombok.Data
@lombok.Builder
@lombok.NoArgsConstructor
@lombok.AllArgsConstructor
class EventMessage {
    private String eventId;
    private String eventType;
    private String payload;
    private Instant timestamp;
}

@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;
}

@lombok.Data
@lombok.Builder
@lombok.NoArgsConstructor
@lombok.AllArgsConstructor
class PaymentEvent {
    private String paymentId;
    private String orderId;
    private java.math.BigDecimal amount;
    private String method;
    private String status;
    private Instant timestamp;
}

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

@lombok.Data
@lombok.Builder
@lombok.NoArgsConstructor
@lombok.AllArgsConstructor
class UserEvent {
    private String userId;
    private String action;
    private String details;
    private Instant timestamp;
}
```

Concurrency Handling

Simple Explanation: Concurrency in Spring Kafka determines how many consumer threads process messages simultaneously. It's critical for optimizing throughput while managing resource usage and maintaining message ordering where needed.

Internal Concurrency Architecture:

```
Kafka Consumer Concurrency Model:

┌─────────────────────────────────────────────────────────────────┐
│                  Topic: orders (6 partitions)                   │
├─────────────────────────────────────────────────────────────────┤
│ Partition 0 │ Partition 1 │ Partition 2 │ Partition 3 │ P4 │ P5 │
└─────────────────────────────────────────────────────────────────┘
                              ↓
┌─────────────────────────────────────────────────────────────────┐
│         ConcurrentMessageListenerContainer (concurrency=3)     │
├─────────────────────────────────────────────────────────────────┤
│  ┌─────────────────┐  ┌─────────────────┐  ┌─────────────────┐ │
│  │   Consumer 1    │  │   Consumer 2    │  │   Consumer 3    │ │
│  │ (Partitions     │  │ (Partitions     │  │ (Partitions     │ │
│  │   0, 1)         │  │   2, 3)         │  │   4, 5)         │ │
│  │                 │  │                 │  │                 │ │
│  │ ┌─────────────┐ │  │ ┌─────────────┐ │  │ ┌─────────────┐ │ │
│  │ │Thread Pool  │ │  │ │Thread Pool  │ │  │ │Thread Pool  │ │ │
│  │ │   (1)       │ │  │ │   (1)       │ │  │ │   (1)       │ │ │
│  │ └─────────────┘ │  │ └─────────────┘ │  │ └─────────────┘ │ │
│  └─────────────────┘  └─────────────────┘  └─────────────────┘ │
└─────────────────────────────────────────────────────────────────┘
```

Advanced Concurrency Configuration and Patterns

```java
import org.springframework.boot.autoconfigure.kafka.KafkaProperties;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.kafka.config.ConcurrentKafkaListenerContainerFactory;
import org.springframework.kafka.core.ConsumerFactory;
import org.springframework.kafka.core.DefaultKafkaConsumerFactory;
import org.springframework.kafka.listener.ContainerProperties;

import org.apache.kafka.clients.consumer.ConsumerConfig;
import org.apache.kafka.common.serialization.StringDeserializer;
import org.springframework.kafka.support.serializer.JsonDeserializer;

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

/**
 * Advanced concurrency configuration for different use cases
 */
@Configuration
public class KafkaConcurrencyConfiguration {

    /**
     * Default concurrency container factory
     * Balanced configuration for general use
     */
    @Bean
    public ConcurrentKafkaListenerContainerFactory<String, Object>
defaultConcurrencyContainerFactory() {
        ConcurrentKafkaListenerContainerFactory<String, Object> factory =
            new ConcurrentKafkaListenerContainerFactory<>();

        factory.setConsumerFactory(defaultConsumerFactory());

        // Moderate concurrency - good balance
        factory.setConcurrency(3);

        // Container properties

factory.getContainerProperties().setAckMode(ContainerProperties.AckMode.MANUAL_IMM
EDIATE);
        factory.getContainerProperties().setPollTimeout(3000);

        return factory;
    }

    /**
     * High concurrency container factory
     * For high-throughput, low-latency scenarios
     */
    @Bean
    public ConcurrentKafkaListenerContainerFactory<String, Object>
highConcurrencyContainerFactory() {
        ConcurrentKafkaListenerContainerFactory<String, Object> factory =
            new ConcurrentKafkaListenerContainerFactory<>();

        // Optimized consumer factory for high throughput
        factory.setConsumerFactory(highThroughputConsumerFactory());

        // High concurrency for maximum throughput
        factory.setConcurrency(8);

        // Optimized container properties

factory.getContainerProperties().setAckMode(ContainerProperties.AckMode.BATCH);
        factory.getContainerProperties().setPollTimeout(1000);
        factory.getContainerProperties().setIdleEventInterval(30000L);

        // Enable batch processing
        factory.setBatchListener(true);

        return factory;
    }

    /**
     * Low concurrency container factory
     * For ordered processing or resource-constrained scenarios
     */
    @Bean
    public ConcurrentKafkaListenerContainerFactory<String, Object>
lowConcurrencyContainerFactory() {
        ConcurrentKafkaListenerContainerFactory<String, Object> factory =
            new ConcurrentKafkaListenerContainerFactory<>();

        factory.setConsumerFactory(orderPreservingConsumerFactory());

        // Single consumer for strict ordering
        factory.setConcurrency(1);

        // Ordered processing settings

factory.getContainerProperties().setAckMode(ContainerProperties.AckMode.RECORD);
        factory.getContainerProperties().setSyncCommits(true);

        return factory;
    }

    /**
     * Dynamic concurrency container factory
     * Adjusts concurrency based on load
     */
    @Bean
    public ConcurrentKafkaListenerContainerFactory<String, Object>
dynamicConcurrencyContainerFactory() {
        ConcurrentKafkaListenerContainerFactory<String, Object> factory =
            new ConcurrentKafkaListenerContainerFactory<>();
        factory.setConsumerFactory(adaptiveConsumerFactory());

        // Start with moderate concurrency
        factory.setConcurrency(4);

        // Enable auto-startup for dynamic adjustment
        factory.setAutoStartup(true);

        return factory;
    }

    // Consumer factory configurations
    @Bean
    public ConsumerFactory<String, Object> defaultConsumerFactory() {
        Map<String, Object> props = new HashMap<>();
        props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        props.put(ConsumerConfig.GROUP_ID_CONFIG, "default-group");
        props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class);
        props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
JsonDeserializer.class);

        // Balanced settings
        props.put(ConsumerConfig.MAX_POLL_RECORDS_CONFIG, 500);
        props.put(ConsumerConfig.SESSION_TIMEOUT_MS_CONFIG, 30000);
        props.put(ConsumerConfig.HEARTBEAT_INTERVAL_MS_CONFIG, 10000);

        return new DefaultKafkaConsumerFactory<>(props);
    }

    @Bean
    public ConsumerFactory<String, Object> highThroughputConsumerFactory() {
        Map<String, Object> props = new HashMap<>();
        props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        props.put(ConsumerConfig.GROUP_ID_CONFIG, "high-throughput-group");
        props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class);
        props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
JsonDeserializer.class);

        // High throughput optimizations
        props.put(ConsumerConfig.MAX_POLL_RECORDS_CONFIG, 1000);
        props.put(ConsumerConfig.FETCH_MIN_BYTES_CONFIG, 1024 * 1024); // 1MB
        props.put(ConsumerConfig.FETCH_MAX_WAIT_MS_CONFIG, 500);
        props.put(ConsumerConfig.RECEIVE_BUFFER_CONFIG, 64 * 1024); // 64KB

        return new DefaultKafkaConsumerFactory<>(props);
    }

    @Bean
    public ConsumerFactory<String, Object> orderPreservingConsumerFactory() {
        Map<String, Object> props = new HashMap<>();
        props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        props.put(ConsumerConfig.GROUP_ID_CONFIG, "ordered-processing-group");
        props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class);
        props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
JsonDeserializer.class);

        // Order preserving settings
        props.put(ConsumerConfig.MAX_POLL_RECORDS_CONFIG, 1);
        props.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, false);
        props.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest");

        return new DefaultKafkaConsumerFactory<>(props);
    }

    @Bean
    public ConsumerFactory<String, Object> adaptiveConsumerFactory() {
        Map<String, Object> props = new HashMap<>();
        props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        props.put(ConsumerConfig.GROUP_ID_CONFIG, "adaptive-group");
        props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class);
        props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
JsonDeserializer.class);

        // Adaptive settings
        props.put(ConsumerConfig.MAX_POLL_RECORDS_CONFIG, 300);
        props.put(ConsumerConfig.SESSION_TIMEOUT_MS_CONFIG, 45000);
        props.put(ConsumerConfig.HEARTBEAT_INTERVAL_MS_CONFIG, 15000);

        return new DefaultKafkaConsumerFactory<>(props);
    }
}

/**
 * Concurrency management service for dynamic adjustment
 */
@Service
@lombok.extern.slf4j.Slf4j
public class ConcurrencyManagementService {

    @Autowired
    private KafkaListenerEndpointRegistry registry;

    private final AtomicInteger currentConcurrency = new AtomicInteger(3);
    private final ConcurrentHashMap<String, Long> lastProcessingTime = new
ConcurrentHashMap<>();

    /**
     * Dynamic concurrency adjustment based on processing metrics
     */
    @Scheduled(fixedDelay = 60000) // Check every minute
    public void adjustConcurrencyBasedOnLoad() {
        Collection<MessageListenerContainer> containers =
registry.getAllListenerContainers();
```

```
        for (MessageListenerContainer container : containers) {
            if (container instanceof ConcurrentMessageListenerContainer
concurrent) {
                adjustContainerConcurrency(concurrent);
            }
        }
    }

    private void adjustContainerConcurrency(ConcurrentMessageListenerContainer
container) {
        String listenerId = container.getListenerId();

        // Get current metrics
        ContainerMetrics metrics = gatherContainerMetrics(container);

        if (metrics.isHighLag() && metrics.isLowCpuUsage()) {
            // Increase concurrency if we have lag and spare CPU
            increaseConcurrency(container, listenerId);
        } else if (metrics.isLowLag() && metrics.isHighCpuUsage()) {
            // Decrease concurrency if no lag but high CPU
            decreaseConcurrency(container, listenerId);
        }
    }

    private void increaseConcurrency(ConcurrentMessageListenerContainer container,
String listenerId) {
        int current = container.getConcurrency();
        int newConcurrency = Math.min(current + 1, 10); // Max 10

        if (newConcurrency > current) {
            log.info("Increasing concurrency for listener {} from {} to {}",
                listenerId, current, newConcurrency);

            container.stop();
            container.setConcurrency(newConcurrency);
            container.start();
        }
    }

    private void decreaseConcurrency(ConcurrentMessageListenerContainer container,
String listenerId) {
        int current = container.getConcurrency();
        int newConcurrency = Math.max(current - 1, 1); // Min 1

        if (newConcurrency < current) {
            log.info("Decreasing concurrency for listener {} from {} to {}",
                listenerId, current, newConcurrency);

            container.stop();
            container.setConcurrency(newConcurrency);
            container.start();
        }
    }
    private ContainerMetrics gatherContainerMetrics(MessageListenerContainer
container) {
        // Implement metrics gathering logic
        // This would typically integrate with micrometer or other monitoring
        return new ContainerMetrics(false, false, false);
    }
}

/**
 * Concurrent processing examples with different patterns
 */
@Component
@lombok.extern.slf4j.Slf4j
public class ConcurrentKafkaListeners {

    /**
     * High concurrency listener for throughput
     * Uses all available cores efficiently
     */
    @KafkaListener(
        topics = "high-volume-events",
        groupId = "high-volume-processor",
        concurrency = "8",
        containerFactory = "highConcurrencyContainerFactory"
    )
    public void handleHighVolumeEvents(
            @Payload List<EventMessage> events,
            @Header(KafkaHeaders.RECEIVED_PARTITION_ID) List<Integer> partitions)
{

        long startTime = System.currentTimeMillis();

        log.info("Processing high-volume batch: {} events across {} partitions",
            events.size(), partitions.stream().distinct().count());

        // Parallel processing within the listener
        events.parallelStream().forEach(this::processEventConcurrently);

        long duration = System.currentTimeMillis() - startTime;
        log.info("Completed high-volume batch processing in {}ms", duration);
    }

    /**
     * Ordered processing listener with single concurrency
     * Maintains strict message ordering
     */
    @KafkaListener(
        topics = "ordered-transactions",
        groupId = "transaction-processor",
        concurrency = "1",
        containerFactory = "lowConcurrencyContainerFactory"
    )
    public void handleOrderedTransactions(@Payload TransactionEvent transaction) {
        log.info("Processing ordered transaction: id={}, sequence={}",
            transaction.getTransactionId(), transaction.getSequenceNumber());

        // Sequential processing to maintain order
        processTransactionInOrder(transaction);
    }

    /**
     * Adaptive concurrency listener
     * Adjusts based on message characteristics
     */
    @KafkaListener(
        topics = "adaptive-processing",
        groupId = "adaptive-processor",
        concurrency = "#{@concurrencyResolver.resolveConcurrency()}",
        containerFactory = "dynamicConcurrencyContainerFactory"
    )
    public void handleAdaptiveProcessing(@Payload ProcessingTask task) {
        log.info("Processing adaptive task: taskId={}, complexity={}",
            task.getTaskId(), task.getComplexity());

        if (task.getComplexity() == TaskComplexity.HIGH) {
            processComplexTaskWithThreadPool(task);
        } else {
            processSimpleTask(task);
        }
    }

    /**
     * Partition-aware concurrent processing
     * Different processing strategies per partition
     */
    @KafkaListener(
        topics = "partition-aware-events",
        groupId = "partition-aware-processor",
        concurrency = "6"
    )
    public void handlePartitionAwareEvents(
            @Payload EventMessage event,
            @Header(KafkaHeaders.RECEIVED_PARTITION_ID) int partition) {

        log.info("Processing event from partition {}: eventId={}",
            partition, event.getEventId());

        // Partition-specific processing logic
        switch (partition % 3) {
            case 0 -> processCriticalPartition(event);
            case 1 -> processNormalPartition(event);
            case 2 -> processBulkPartition(event);
        }
    }

    // Processing methods
    private void processEventConcurrently(EventMessage event) {
        log.debug("Processing event concurrently: {}", event.getEventId());
        // Simulate processing time
        try {
            Thread.sleep(10);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }

    private void processTransactionInOrder(TransactionEvent transaction) {
        log.debug("Processing transaction in order: {}",
transaction.getTransactionId());
    }

    private void processComplexTaskWithThreadPool(ProcessingTask task) {
        log.debug("Processing complex task with thread pool: {}",
task.getTaskId());
        // Use internal thread pool for CPU-intensive tasks
    }

    private void processSimpleTask(ProcessingTask task) {
        log.debug("Processing simple task: {}", task.getTaskId());
    }

    private void processCriticalPartition(EventMessage event) {
        log.debug("Processing critical partition event: {}", event.getEventId());
    }

    private void processNormalPartition(EventMessage event) {
        log.debug("Processing normal partition event: {}", event.getEventId());
    }

    private void processBulkPartition(EventMessage event) {
        log.debug("Processing bulk partition event: {}", event.getEventId());
    }
}

/**
 * Concurrency resolver for dynamic concurrency
 */
@Component
public class ConcurrencyResolver {

    public int resolveConcurrency() {
        // Dynamic concurrency based on current system load
        int availableProcessors = Runtime.getRuntime().availableProcessors();
        int systemLoad = getCurrentSystemLoad();

        if (systemLoad < 50) {
            return Math.min(availableProcessors, 8);
        } else if (systemLoad < 80) {
            return Math.min(availableProcessors / 2, 4);
        } else {
            return 2;
        }
    }

    private int getCurrentSystemLoad() {
        // Implement system load detection
        return 30; // Placeholder
    }
}

// Supporting classes
@lombok.Data
@lombok.AllArgsConstructor
class ContainerMetrics {
    private boolean highLag;
    private boolean lowLag;
    private boolean lowCpuUsage;
    private boolean highCpuUsage;

    public ContainerMetrics(boolean highLag, boolean lowCpuUsage, boolean
highCpuUsage) {
        this.highLag = highLag;
        this.lowLag = !highLag;
        this.lowCpuUsage = lowCpuUsage;
        this.highCpuUsage = highCpuUsage;
    }
}

@lombok.Data
@lombok.Builder
@lombok.NoArgsConstructor
@lombok.AllArgsConstructor
class TransactionEvent {
    private String transactionId;
    private Long sequenceNumber;
    private java.math.BigDecimal amount;
    private String type;
    private Instant timestamp;
}

@lombok.Data
@lombok.Builder
@lombok.NoArgsConstructor
@lombok.AllArgsConstructor
class ProcessingTask {
    private String taskId;
    private TaskComplexity complexity;
    private String payload;
    private Instant timestamp;
}

enum TaskComplexity {
    LOW, MEDIUM, HIGH
}
```

Message Filtering Strategies

Simple Explanation: Message filtering allows consumers to selectively process only specific messages that meet certain criteria, reducing processing overhead and enabling focused business logic implementation.

Why Message Filtering is Important:

- **Resource Optimization**: Avoid processing irrelevant messages
- **Business Logic Separation**: Different consumers handle different message types
- **Performance**: Reduce CPU and memory usage by early filtering
- **Maintainability**: Clear separation of concerns in microservices

Advanced Message Filtering Implementation

```java
import org.springframework.kafka.listener.adapter.RecordFilterStrategy;
import org.springframework.kafka.annotation.KafkaListener;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

import org.apache.kafka.clients.consumer.ConsumerRecord;

/**
 * Advanced message filtering strategies and implementations
 */
@Configuration
public class MessageFilteringConfiguration {

    /**
     * Simple content-based filter
     * Filters messages based on payload content
     */
    @Bean("contentFilter")
    public RecordFilterStrategy<String, Object> contentBasedFilter() {
        return new RecordFilterStrategy<String, Object>() {
            @Override
            public boolean filter(ConsumerRecord<String, Object> consumerRecord) {
                Object value = consumerRecord.value();

                if (value instanceof EventMessage event) {
                    // Filter out test messages
                    return event.getEventType().startsWith("TEST_");
                }

                return false; // Don't filter by default
            }
        };
    }

    /**
     * Header-based filter
     * Filters based on message headers
     */
    @Bean("headerFilter")
    public RecordFilterStrategy<String, Object> headerBasedFilter() {
        return consumerRecord -> {
            // Filter based on headers
            String environment = getHeaderValue(consumerRecord, "environment");
            String messageType = getHeaderValue(consumerRecord, "message-type");

            // Filter out development environment messages in production
            if ("development".equals(environment)) {
                return true;
            }

            // Filter out specific message types
            return "debug".equals(messageType) || "trace".equals(messageType);
        };
    }

    /**
     * Business rule-based filter
     * Complex filtering based on business logic
     */
    @Bean("businessRuleFilter")
    public RecordFilterStrategy<String, Object> businessRuleBasedFilter() {
        return consumerRecord -> {
            Object value = consumerRecord.value();

            if (value instanceof OrderEvent order) {
                // Filter out orders below minimum amount
                return order.getAmount().compareTo(new
java.math.BigDecimal("10.00")) < 0;
            }

            if (value instanceof PaymentEvent payment) {
                // Filter out failed payments
                return "FAILED".equals(payment.getStatus());
            }

            return false;
        };
    }

    /**
     * Geographic filter
     * Filter based on geographic regions
     */
    @Bean("geographicFilter")
    public RecordFilterStrategy<String, Object> geographicFilter() {
        return consumerRecord -> {
            String region = getHeaderValue(consumerRecord, "region");
            String topic = consumerRecord.topic();

            // US consumer only processes US region messages
            if (topic.contains("us-consumer")) {
                return !"US".equals(region);
            }

            // EU consumer only processes EU region messages
            if (topic.contains("eu-consumer")) {
                return !"EU".equals(region);
            }

            return false;
        };
    }

    /**
     * Time-based filter
     * Filter based on message timestamps
     */
    @Bean("timeBasedFilter")
    public RecordFilterStrategy<String, Object> timeBasedFilter() {
        return consumerRecord -> {
            long messageTimestamp = consumerRecord.timestamp();
            long currentTime = System.currentTimeMillis();

            // Filter out messages older than 1 hour
            long oneHourAgo = currentTime - (60 * 60 * 1000);

            return messageTimestamp < oneHourAgo;
        };
    }

    /**
     * Composite filter combining multiple strategies
     */
    @Bean("compositeFilter")
    public RecordFilterStrategy<String, Object> compositeFilter() {
        return consumerRecord -> {
            // Apply multiple filter strategies
            if (contentBasedFilter().filter(consumerRecord)) return true;
            if (headerBasedFilter().filter(consumerRecord)) return true;
            if (businessRuleFilter().filter(consumerRecord)) return true;

            return false;
        };
    }

    /**
     * Dynamic filter that can be configured at runtime
     */
    @Bean("dynamicFilter")
    public RecordFilterStrategy<String, Object> dynamicFilter() {
        return new DynamicRecordFilterStrategy();
    }

    // Helper method
    private String getHeaderValue(ConsumerRecord<String, Object> record, String
headerName) {
        if (record.headers() != null) {
            org.apache.kafka.common.header.Header header =
record.headers().lastHeader(headerName);
            if (header != null) {
                return new String(header.value());
            }
        }
        return null;
    }
}

/**
 * Dynamic filter strategy that can be configured at runtime
 */
public class DynamicRecordFilterStrategy implements RecordFilterStrategy<String,
Object> {

    private volatile FilterConfiguration filterConfig = new FilterConfiguration();

    @Override
    public boolean filter(ConsumerRecord<String, Object> consumerRecord) {
        FilterConfiguration config = filterConfig; // Volatile read

        // Apply configured filters
        if (config.isContentFilterEnabled() &&
shouldFilterByContent(consumerRecord, config)) {
            return true;
        }

        if (config.isHeaderFilterEnabled() &&
shouldFilterByHeaders(consumerRecord, config)) {
            return true;
        }

        if (config.isBusinessRuleFilterEnabled() &&
shouldFilterByBusinessRule(consumerRecord, config)) {
            return true;
        }

        return false;
    }

    public void updateFilterConfiguration(FilterConfiguration newConfig) {
        this.filterConfig = newConfig; // Volatile write
    }

    private boolean shouldFilterByContent(ConsumerRecord<String, Object> record,
FilterConfiguration config) {
        Object value = record.value();

        for (String excludedType : config.getExcludedEventTypes()) {
            if (value instanceof EventMessage event &&
excludedType.equals(event.getEventType())) {
                return true;
            }
        }

        return false;
    }

    private boolean shouldFilterByHeaders(ConsumerRecord<String, Object> record,
FilterConfiguration config) {
        for (Map.Entry<String, String> excludedHeader :
config.getExcludedHeaders().entrySet()) {
            String headerValue = getHeaderValue(record, excludedHeader.getKey());
            if (excludedHeader.getValue().equals(headerValue)) {
                return true;
            }
        }

        return false;
    }

    private boolean shouldFilterByBusinessRule(ConsumerRecord<String, Object>
record, FilterConfiguration config) {
        Object value = record.value();

        if (value instanceof OrderEvent order) {
            return order.getAmount().compareTo(config.getMinOrderAmount()) < 0;
        }

        return false;
    }

    private String getHeaderValue(ConsumerRecord<String, Object> record, String
headerName) {
        if (record.headers() != null) {
            org.apache.kafka.common.header.Header header =
record.headers().lastHeader(headerName);
            if (header != null) {
                return new String(header.value());
            }
        }
        return null;
    }
}

/**
 * Listener examples using different filtering strategies
 */
@Component
@lombok.extern.slf4j.Slf4j
public class FilteredKafkaListeners {

    /**
     * Content-filtered listener
     * Only processes non-test events
     */
    @KafkaListener(
        topics = "all-events",
        groupId = "production-events-processor",
        containerFactory = "contentFilteredContainerFactory",
        filter = "contentFilter"
    )
    public void handleProductionEvents(@Payload EventMessage event) {
        log.info("Processing production event: type={}, id={}",
            event.getEventType(), event.getEventId());

        processProductionEvent(event);
    }

    /**
     * Header-filtered listener
     * Filters based on environment and message type headers
     */
    @KafkaListener(
        topics = "system-events",
        groupId = "production-system-processor",
        containerFactory = "headerFilteredContainerFactory",
        filter = "headerFilter"
    )
    public void handleProductionSystemEvents(
            @Payload Object event,
            @Header(KafkaHeaders.RECEIVED_TOPIC) String topic,
            @Header(value = "environment", required = false) String environment) {

        log.info("Processing production system event from environment: {}",
environment);
        processSystemEvent(event);
    }

    /**
     * Business rule filtered listener
     * Only processes orders above minimum amount
     */
    @KafkaListener(
        topics = "order-events",
        groupId = "significant-orders-processor",
        filter = "businessRuleFilter"
    )
    public void handleSignificantOrders(@Payload OrderEvent order) {
        log.info("Processing significant order: orderId={}, amount={}",
            order.getOrderId(), order.getAmount());

        processSignificantOrder(order);
    }

    /**
     * Geographic filtered listener
     * Region-specific processing
     */
    @KafkaListener(
        topics = "global-user-events",
        groupId = "us-user-processor",
        filter = "geographicFilter"
    )
    public void handleUSUserEvents(
            @Payload UserEvent userEvent,
            @Header(value = "region", required = false) String region) {

        log.info("Processing US user event: userId={}, region={}",
            userEvent.getUserId(), region);

        processUSUserEvent(userEvent);
    }

    /**
     * Time-filtered listener
     * Only processes recent messages
     */
    @KafkaListener(
        topics = "time-sensitive-events",
        groupId = "recent-events-processor",
        filter = "timeBasedFilter"
    )
    public void handleRecentEvents(@Payload EventMessage event) {
        log.info("Processing recent event: eventId={}, type={}",
            event.getEventId(), event.getEventType());

        processRecentEvent(event);
    }

    /**
     * Composite filtered listener
     * Multiple filtering strategies combined
     */
    @KafkaListener(
        topics = "comprehensive-events",
        groupId = "filtered-processor",
        filter = "compositeFilter"
    )
    public void handleComprehensivelyFilteredEvents(@Payload Object event) {
        log.info("Processing comprehensively filtered event: {}",
event.getClass().getSimpleName());

        processFilteredEvent(event);
    }

    /**
     * Dynamic filtered listener
     * Runtime configurable filtering
     */
    @KafkaListener(
        topics = "configurable-events",
        groupId = "dynamic-processor",
        filter = "dynamicFilter"
```

```java
    )
    public void handleDynamicallyFilteredEvents(@Payload EventMessage event) {
        log.info("Processing dynamically filtered event: type={}, id={}",
            event.getEventType(), event.getEventId());

        processDynamicEvent(event);
    }

    // Business logic methods
    private void processProductionEvent(EventMessage event) {
        log.debug("Processing production event logic: {}", event.getEventId());
    }

    private void processSystemEvent(Object event) {
        log.debug("Processing system event logic: {}",
event.getClass().getSimpleName());
    }

    private void processSignificantOrder(OrderEvent order) {
        log.debug("Processing significant order logic: {}", order.getOrderId());
    }

    private void processUSUserEvent(UserEvent userEvent) {
        log.debug("Processing US user event logic: {}", userEvent.getUserId());
    }

    private void processRecentEvent(EventMessage event) {
        log.debug("Processing recent event logic: {}", event.getEventId());
    }

    private void processFilteredEvent(Object event) {
        log.debug("Processing filtered event logic: {}",
event.getClass().getSimpleName());
    }

    private void processDynamicEvent(EventMessage event) {
        log.debug("Processing dynamic event logic: {}", event.getEventId());
    }
}

/**
 * Container factory configurations for different filters
 */
@Configuration
public class FilteredContainerFactoryConfiguration {

    @Bean
    public ConcurrentKafkaListenerContainerFactory<String, Object>
contentFilteredContainerFactory() {
        ConcurrentKafkaListenerContainerFactory<String, Object> factory =
            new ConcurrentKafkaListenerContainerFactory<>();

        factory.setConsumerFactory(defaultConsumerFactory());
        factory.setRecordFilterStrategy(contentBasedFilter());
        factory.setConcurrency(3);

        return factory;
    }

    @Bean
    public ConcurrentKafkaListenerContainerFactory<String, Object>
headerFilteredContainerFactory() {
        ConcurrentKafkaListenerContainerFactory<String, Object> factory =
            new ConcurrentKafkaListenerContainerFactory<>();

        factory.setConsumerFactory(defaultConsumerFactory());
        factory.setRecordFilterStrategy(headerBasedFilter());
        factory.setConcurrency(2);

        return factory;
    }

    // Placeholder methods - would be implemented in actual configuration
    private ConsumerFactory<String, Object> defaultConsumerFactory() {
        return null; // Implementation would be here
    }

    private RecordFilterStrategy<String, Object> contentBasedFilter() {
        return null; // Implementation would be here
    }

    private RecordFilterStrategy<String, Object> headerBasedFilter() {
        return null; // Implementation would be here
    }
}

/**
 * Filter configuration management service
 */
@Service
@lombok.extern.slf4j.Slf4j
public class FilterConfigurationService {

    @Autowired
    private DynamicRecordFilterStrategy dynamicFilter;

    /**
     * Update filter configuration at runtime
     */
    public void updateFilterConfiguration(String eventTypesToExclude,
                                        String minOrderAmount) {

        FilterConfiguration newConfig = FilterConfiguration.builder()
            .contentFilterEnabled(true)
            .businessRuleFilterEnabled(true)
            .excludedEventTypes(Set.of(eventTypesToExclude.split(",")))
            .minOrderAmount(new java.math.BigDecimal(minOrderAmount))
            .build();
        dynamicFilter.updateFilterConfiguration(newConfig);

        log.info("Updated filter configuration: excludedTypes={}, minAmount={}",
            eventTypesToExclude, minOrderAmount);
    }

    /**
     * Enable/disable specific filters
     */
    public void toggleFilter(String filterType, boolean enabled) {
        // Implementation for toggling specific filters
        log.info("Toggled filter {}: {}", filterType, enabled);
    }
}

// Configuration classes
@lombok.Data
@lombok.Builder
@lombok.NoArgsConstructor
@lombok.AllArgsConstructor
class FilterConfiguration {
    private boolean contentFilterEnabled = true;
    private boolean headerFilterEnabled = true;
    private boolean businessRuleFilterEnabled = true;

    private Set<String> excludedEventTypes = new HashSet<>();
    private Map<String, String> excludedHeaders = new HashMap<>();
    private java.math.BigDecimal minOrderAmount = java.math.BigDecimal.ZERO;

    // Time-based filtering
    private long maxMessageAgeMs = 3600000; // 1 hour

    // Geographic filtering
    private Set<String> allowedRegions = new HashSet<>();
}
```

🏗 Listener Containers

What are Listener Containers?

Simple Explanation: Listener containers are the runtime components in Spring Kafka that manage the lifecycle of Kafka consumers, handle the polling loop, manage partition assignment, coordinate consumer groups, and provide error handling and recovery mechanisms.

Internal Container Architecture:

```
Listener Container Hierarchy:

┌─────────────────────────────────────────────────────────────────┐
```

```
│              AbstractMessageListenerContainer              │
├─────────────────────────────────────────────────────────────┤
│  ┌─────────────────┐  ┌─────────────────┐  ┌─────────────────┐ │
│  │   Lifecycle     │  │ Error Handling  │  │  Acknowledgment │ │
│  │  Management     │  │   & Retry       │  │   Management    │ │
│  └─────────────────┘  └─────────────────┘  └─────────────────┘ │
└─────────────────────────────────────────────────────────────┘
                              ↓
┌─────────────────────────────────────────────────────────────┐
│          ConcurrentMessageListenerContainer               │
├─────────────────────────────────────────────────────────────┤
│  ┌─────────────────┐  ┌─────────────────┐  ┌─────────────────┐ │
│  │KafkaMessage     │  │KafkaMessage     │  │KafkaMessage     │ │
│  │ListenerContainer│  │ListenerContainer│  │ListenerContainer│ │
│  │   (Thread 1)    │  │   (Thread 2)    │  │   (Thread 3)    │ │
│  └─────────────────┘  └─────────────────┘  └─────────────────┘ │
└─────────────────────────────────────────────────────────────┘
                              ↓
┌─────────────────────────────────────────────────────────────┐
│               KafkaMessageListenerContainer                 │
├─────────────────────────────────────────────────────────────┤
│  ┌─────────────────┐  ┌─────────────────┐  ┌─────────────────┐ │
│  │  Consumer Poll  │  │  Message Proc.  │  │  Offset Commit  │ │
│  │     Loop        │  │    Thread       │  │   Management    │ │
│  └─────────────────┘  └─────────────────┘  └─────────────────┘ │
└─────────────────────────────────────────────────────────────┘
```

ConcurrentMessageListenerContainer

Advanced Container Configuration and Management

```java
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.kafka.config.ConcurrentKafkaListenerContainerFactory;
import org.springframework.kafka.config.KafkaListenerEndpointRegistry;
import org.springframework.kafka.core.ConsumerFactory;
import org.springframework.kafka.listener.*;
import org.springframework.kafka.support.TopicPartitionOffset;

import org.apache.kafka.clients.consumer.ConsumerConfig;
import org.apache.kafka.common.TopicPartition;

import java.time.Duration;
import java.util.Collection;
import java.util.List;
import java.util.concurrent.CountDownLatch;

/**
 * Advanced ConcurrentMessageListenerContainer configuration
 */
@Configuration
```

```
@lombok.extern.slf4j.Slf4j
public class ConcurrentContainerConfiguration {

    /**
     * Default concurrent container factory
     */
    @Bean
    public ConcurrentKafkaListenerContainerFactory<String, Object>
concurrentContainerFactory() {
        ConcurrentKafkaListenerContainerFactory<String, Object> factory =
            new ConcurrentKafkaListenerContainerFactory<>();

        factory.setConsumerFactory(consumerFactory());

        // Concurrency settings
        factory.setConcurrency(4);
        factory.setAutoStartup(true);

        // Container properties
        ContainerProperties containerProperties =
factory.getContainerProperties();

containerProperties.setAckMode(ContainerProperties.AckMode.MANUAL_IMMEDIATE);
        containerProperties.setPollTimeout(3000);
        containerProperties.setIdleEventInterval(30000L);

        // Error handling
        factory.setCommonErrorHandler(defaultErrorHandler());

        // Consumer lifecycle
        factory.setContainerCustomizer(this::customizeContainer);

        return factory;
    }

    /**
     * High-performance concurrent container
     */
    @Bean
    public ConcurrentKafkaListenerContainerFactory<String, Object>
highPerformanceConcurrentFactory() {
        ConcurrentKafkaListenerContainerFactory<String, Object> factory =
            new ConcurrentKafkaListenerContainerFactory<>();

        factory.setConsumerFactory(highPerformanceConsumerFactory());

        // Maximum concurrency for high throughput
        factory.setConcurrency(8);

        // Optimized container properties
        ContainerProperties props = factory.getContainerProperties();
        props.setPollTimeout(1000);
        props.setIdleEventInterval(10000L);
        props.setMonitorInterval(5);
        props.setNoPollThreshold(3.0f);

        // Batch processing
        factory.setBatchListener(false); // Single record processing for lower
latency

        return factory;
    }

    /**
     * Resilient concurrent container with advanced error handling
     */
    @Bean
    public ConcurrentKafkaListenerContainerFactory<String, Object>
resilientConcurrentFactory() {
        ConcurrentKafkaListenerContainerFactory<String, Object> factory =
            new ConcurrentKafkaListenerContainerFactory<>();

        factory.setConsumerFactory(consumerFactory());
        factory.setConcurrency(3);

        // Advanced error handling
        factory.setCommonErrorHandler(resilientErrorHandler());

        // Container properties for resilience
        ContainerProperties props = factory.getContainerProperties();
        props.setAckMode(ContainerProperties.AckMode.MANUAL);
        props.setSyncCommits(true);
        props.setCommitCallback(commitCallback());

        // Graceful shutdown
        props.setShutdownTimeout(Duration.ofSeconds(30));

        return factory;
    }

    /**
     * Custom container customization
     */
    private void customizeContainer(AbstractMessageListenerContainer<String,
Object> container) {
        // Add custom listeners
        container.setupMessageListener(new LoggingMessageListener());

        // Set custom thread name prefix
        if (container instanceof ConcurrentMessageListenerContainer<?> concurrent)
{
            // Additional concurrent-specific customization
            log.info("Customizing concurrent container with {} consumers",
                concurrent.getConcurrency());
        }
    }

    /**
```

```
     * Container lifecycle management service
     */
    @Component
    public static class ContainerLifecycleManager {

        @Autowired
        private KafkaListenerEndpointRegistry registry;

        /**
         * Start specific container
         */
        public void startContainer(String listenerId) {
            MessageListenerContainer container =
registry.getListenerContainer(listenerId);
            if (container != null && !container.isRunning()) {
                container.start();
                log.info("Started container: {}", listenerId);
            }
        }

        /**
         * Stop specific container
         */
        public void stopContainer(String listenerId) {
            MessageListenerContainer container =
registry.getListenerContainer(listenerId);
            if (container != null && container.isRunning()) {
                container.stop();
                log.info("Stopped container: {}", listenerId);
            }
        }

        /**
         * Restart container with new configuration
         */
        public void restartContainer(String listenerId) {
            MessageListenerContainer container =
registry.getListenerContainer(listenerId);
            if (container != null) {
                container.stop();
                // Wait for graceful shutdown
                try {
                    Thread.sleep(5000);
                } catch (InterruptedException e) {
                    Thread.currentThread().interrupt();
                }
                container.start();
                log.info("Restarted container: {}", listenerId);
            }
        }

        /**
         * Get container metrics
         */
        public ContainerMetrics getContainerMetrics(String listenerId) {
            MessageListenerContainer container =
registry.getListenerContainer(listenerId);
            if (container instanceof ConcurrentMessageListenerContainer<?>
concurrent) {
                return ContainerMetrics.builder()
                    .listenerId(listenerId)
                    .running(container.isRunning())
                    .concurrency(concurrent.getConcurrency())
                    .activeContainers(getActiveContainerCount(concurrent))
                    .build();
            }
            return null;
        }

        /**
         * Pause container temporarily
         */
        public void pauseContainer(String listenerId) {
            MessageListenerContainer container =
registry.getListenerContainer(listenerId);
            if (container != null) {
                container.pause();
                log.info("Paused container: {}", listenerId);
            }
        }

        /**
         * Resume paused container
         */
        public void resumeContainer(String listenerId) {
            MessageListenerContainer container =
registry.getListenerContainer(listenerId);
            if (container != null) {
                container.resume();
                log.info("Resumed container: {}", listenerId);
            }
        }

        /**
         * Scale container concurrency dynamically
         */
        public void scaleContainer(String listenerId, int newConcurrency) {
            MessageListenerContainer container =
registry.getListenerContainer(listenerId);
            if (container instanceof ConcurrentMessageListenerContainer<?>
concurrent) {
                concurrent.stop();
                concurrent.setConcurrency(newConcurrency);
                concurrent.start();
                log.info("Scaled container {} to concurrency: {}", listenerId,
newConcurrency);
            }
        }
        private int getActiveContainerCount(ConcurrentMessageListenerContainer<?>
container) {
            // Implementation would check active containers
            return container.getConcurrency();
        }
    }

    /**
     * Container health monitoring service
     */
    @Component
    public static class ContainerHealthMonitor {

        @Autowired
        private KafkaListenerEndpointRegistry registry;

        @Scheduled(fixedDelay = 60000) // Check every minute
        public void monitorContainerHealth() {
            Collection<MessageListenerContainer> containers =
registry.getAllListenerContainers();

            for (MessageListenerContainer container : containers) {
                checkContainerHealth(container);
            }
        }

        private void checkContainerHealth(MessageListenerContainer container) {
            String listenerId = container.getListenerId();

            if (!container.isRunning()) {
                log.warn("Container {} is not running", listenerId);
                alertContainerDown(listenerId);
                return;
            }

            if (container instanceof ConcurrentMessageListenerContainer<?>
concurrent) {
                checkConcurrentContainerHealth(concurrent);
            }
        }

        private void
checkConcurrentContainerHealth(ConcurrentMessageListenerContainer<?> container) {
            String listenerId = container.getListenerId();

            // Check if any child containers have stopped
            if (container.isRunning()) {
                log.debug("Concurrent container {} is healthy", listenerId);
            } else {
                log.error("Concurrent container {} has issues", listenerId);
                attemptContainerRecovery(container);
            }
        }
        private void alertContainerDown(String listenerId) {
            log.error("🚨 ALERT: Container {} is down", listenerId);
            // Could integrate with alerting system
        }

        private void attemptContainerRecovery(MessageListenerContainer container)
{
            log.info("Attempting to recover container: {}",
container.getListenerId());
            try {
                container.stop();
                Thread.sleep(5000);
                container.start();
                log.info("Successfully recovered container: {}",
container.getListenerId());
            } catch (Exception e) {
                log.error("Failed to recover container: {}",
container.getListenerId(), e);
            }
        }
    }

    // Helper methods and configurations
    private ConsumerFactory<String, Object> consumerFactory() {
        // Implementation would be here
        return null;
    }

    private ConsumerFactory<String, Object> highPerformanceConsumerFactory() {
        // Implementation would be here with optimized settings
        return null;
    }

    private CommonErrorHandler defaultErrorHandler() {
        return new DefaultErrorHandler();
    }

    private CommonErrorHandler resilientErrorHandler() {
        return new DefaultErrorHandler(
            new DeadLetterPublishingRecoverer(kafkaTemplate()),
            new org.springframework.util.backoff.FixedBackOff(1000L, 3L)
        );
    }

    private OffsetCommitCallback commitCallback() {
        return (offsets, exception) -> {
            if (exception != null) {
                log.error("Commit failed for offsets: {}", offsets, exception);
            } else {
                log.debug("Successfully committed offsets: {}", offsets);
            }
        };
    }
    private Object kafkaTemplate() {
        // Implementation would return KafkaTemplate
        return null;
    }
}

/**
 * Advanced concurrent listener implementations
 */
@Component
@lombok.extern.slf4j.Slf4j
public class ConcurrentMessageListeners {

    @Autowired
    private ContainerLifecycleManager lifecycleManager;

    /**
     * High-performance concurrent listener
     */
    @KafkaListener(
        id = "high-performance-listener",
        topics = "high-volume-topic",
        groupId = "high-performance-group",
        concurrency = "8",
        containerFactory = "highPerformanceConcurrentFactory"
    )
    public void handleHighPerformanceMessages(@Payload EventMessage message) {
        long startTime = System.nanoTime();

        try {
            processHighPerformanceMessage(message);

            long duration = System.nanoTime() - startTime;
            if (duration > 1_000_000) { // Log if processing takes > 1ms
                log.debug("High-performance processing took {}μs for event: {}",
                    duration / 1000, message.getEventId());
            }

        } catch (Exception e) {
            log.error("High-performance processing failed for event: {}",
                message.getEventId(), e);
            throw e;
        }
    }

    /**
     * Resilient concurrent listener with comprehensive error handling
     */
    @KafkaListener(
        id = "resilient-listener",
        topics = "critical-topic",
        groupId = "resilient-group",
        concurrency = "3",
        containerFactory = "resilientConcurrentFactory"
    )
    public void handleResilientMessages(@Payload CriticalMessage message,
                                        Acknowledgment ack,
                                        Consumer<String, CriticalMessage> consumer)
{

        log.info("Processing critical message: id={}, priority={}",
            message.getMessageId(), message.getPriority());

        try {
            processCriticalMessage(message);

            // Manual acknowledgment after successful processing
            ack.acknowledge();

            log.info("Successfully processed critical message: {}",
message.getMessageId());

        } catch (Exception e) {
            log.error("Failed to process critical message: {}",
message.getMessageId(), e);

            // Don't acknowledge - will be retried
            handleCriticalProcessingFailure(message, e);
            throw e;
        }
    }

    /**
     * Dynamic concurrent listener that can be scaled at runtime
     */
    @KafkaListener(
        id = "scalable-listener",
        topics = "scalable-topic",
        groupId = "scalable-group",
        concurrency = "4",
        containerFactory = "concurrentContainerFactory"
    )
    public void handleScalableMessages(@Payload ScalableMessage message) {
        log.info("Processing scalable message: id={}, workload={}",
            message.getMessageId(), message.getWorkloadType());

        try {
            processScalableMessage(message);

            // Trigger scaling based on workload
            if (shouldScaleUp(message)) {
                triggerScaleUp();
            } else if (shouldScaleDown(message)) {
                triggerScaleDown();
            }

        } catch (Exception e) {
            log.error("Failed to process scalable message: {}", 
message.getMessageId(), e);
            throw e;
        }
    }

    /**
     * Container with custom partition assignment
     */
    @KafkaListener(
        id = "partition-specific-listener",
        topicPartitions = {
            @TopicPartition(topic = "partitioned-topic", partitions = {"0", "1"}),
            @TopicPartition(topic = "partitioned-topic",
                partitions = "2",
                partitionOffsets = @PartitionOffset(partition = "2", initialOffset
= "100"))
        },
        groupId = "partition-specific-group",
        concurrency = "3"
    )
    public void handlePartitionSpecificMessages(
            @Payload PartitionedMessage message,
            @Header(KafkaHeaders.RECEIVED_PARTITION_ID) int partition,
            @Header(KafkaHeaders.OFFSET) long offset) {

        log.info("Processing partitioned message: partition={}, offset={}, id={}",
            partition, offset, message.getMessageId());

        // Partition-specific processing logic
        processPartitionedMessage(message, partition);
    }

    // Business logic methods
    private void processHighPerformanceMessage(EventMessage message) {
        // High-performance processing logic
        log.debug("High-performance processing: {}", message.getEventId());
    }

    private void processCriticalMessage(CriticalMessage message) {
        // Critical message processing logic
        log.debug("Critical processing: {}", message.getMessageId());

        if ("CRITICAL".equals(message.getPriority())) {
            // Special handling for critical priority
            processCriticalPriorityMessage(message);
        }
    }

    private void processCriticalPriorityMessage(CriticalMessage message) {
        log.debug("Critical priority processing: {}", message.getMessageId());
    }

    private void handleCriticalProcessingFailure(CriticalMessage message,
Exception e) {
        log.error("Handling critical processing failure for: {}",
message.getMessageId());
        // Could send alerts, update monitoring, etc.
    }

    private void processScalableMessage(ScalableMessage message) {
        log.debug("Scalable processing: {}", message.getMessageId());
    }

    private boolean shouldScaleUp(ScalableMessage message) {
        return "HIGH_LOAD".equals(message.getWorkloadType());
    }

    private boolean shouldScaleDown(ScalableMessage message) {
        return "LOW_LOAD".equals(message.getWorkloadType());
    }

    private void triggerScaleUp() {
        log.info("Triggering scale up operation");
        lifecycleManager.scaleContainer("scalable-listener", 6);
    }

    private void triggerScaleDown() {
        log.info("Triggering scale down operation");
        lifecycleManager.scaleContainer("scalable-listener", 2);
    }

    private void processPartitionedMessage(PartitionedMessage message, int
partition) {
        log.debug("Partitioned processing: partition={}, id={}", partition,
message.getMessageId());
    }
}

// Supporting classes
@lombok.Data
@lombok.Builder
@lombok.NoArgsConstructor
@lombok.AllArgsConstructor
class ContainerMetrics {
    private String listenerId;
    private boolean running;
    private int concurrency;
    private int activeContainers;
}

@lombok.Data
@lombok.Builder
@lombok.NoArgsConstructor
@lombok.AllArgsConstructor
class CriticalMessage {
    private String messageId;
    private String priority;
    private String content;
    private java.time.Instant timestamp;
}

@lombok.Data
@lombok.Builder
@lombok.NoArgsConstructor
@lombok.AllArgsConstructor
class ScalableMessage {
    private String messageId;
    private String workloadType;
    private String content;
    private java.time.Instant timestamp;
}

@lombok.Data
@lombok.Builder
@lombok.NoArgsConstructor
@lombok.AllArgsConstructor
class PartitionedMessage {
    private String messageId;
    private String content;
    private java.time.Instant timestamp;
}

/**
 * Custom message listener for logging
 */
class LoggingMessageListener implements ConsumerAwareListenerErrorHandler {

    @Override
    public Object handleError(Message<?> message, ListenerExecutionFailedException
exception,
                              Consumer<?, ?> consumer) {
        log.error("Message processing failed: {}", message, exception);
        return null;
    }
}
```

BatchMessageListenerContainer

Advanced Batch Processing Configuration

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.kafka.config.ConcurrentKafkaListenerContainerFactory;
import org.springframework.kafka.listener.BatchErrorHandler;
import org.springframework.kafka.listener.BatchMessageListener;
import org.springframework.kafka.listener.ContainerProperties;

/**
```

```
  * Advanced batch message processing configuration
  */
@Configuration
@lombok.extern.slf4j.Slf4j
public class BatchProcessingConfiguration {

    /**
     * Batch processing container factory
     */
    @Bean
    public ConcurrentKafkaListenerContainerFactory<String, Object>
batchContainerFactory() {
        ConcurrentKafkaListenerContainerFactory<String, Object> factory =
            new ConcurrentKafkaListenerContainerFactory<>();

        factory.setConsumerFactory(batchConsumerFactory());

        // Enable batch processing
        factory.setBatchListener(true);
        factory.setConcurrency(4);

        // Batch-specific container properties
        ContainerProperties props = factory.getContainerProperties();
        props.setAckMode(ContainerProperties.AckMode.BATCH);
        props.setPollTimeout(5000);

        // Batch error handling
        factory.setBatchErrorHandler(batchErrorHandler());

        return factory;
    }

    /**
     * High-throughput batch container
     */
    @Bean
    public ConcurrentKafkaListenerContainerFactory<String, Object>
highThroughputBatchFactory() {
        ConcurrentKafkaListenerContainerFactory<String, Object> factory =
            new ConcurrentKafkaListenerContainerFactory<>();

        factory.setConsumerFactory(highThroughputBatchConsumerFactory());

        factory.setBatchListener(true);
        factory.setConcurrency(6);

        // Optimized for high throughput
        ContainerProperties props = factory.getContainerProperties();
        props.setAckMode(ContainerProperties.AckMode.BATCH);
        props.setPollTimeout(1000);
        props.setIdleEventInterval(10000L);

        return factory;
    }
    /**
     * Ordered batch processing container
     */
    @Bean
    public ConcurrentKafkaListenerContainerFactory<String, Object>
orderedBatchFactory() {
        ConcurrentKafkaListenerContainerFactory<String, Object> factory =
            new ConcurrentKafkaListenerContainerFactory<>();

        factory.setConsumerFactory(orderedBatchConsumerFactory());

        factory.setBatchListener(true);
        factory.setConcurrency(1); // Single consumer for ordering

        // Ordered processing settings
        ContainerProperties props = factory.getContainerProperties();
        props.setAckMode(ContainerProperties.AckMode.BATCH);
        props.setSyncCommits(true);

        return factory;
    }

    private ConsumerFactory<String, Object> batchConsumerFactory() {
        Map<String, Object> props = new HashMap<>();
        props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        props.put(ConsumerConfig.GROUP_ID_CONFIG, "batch-group");
        props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class);
        props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
JsonDeserializer.class);

        // Batch optimization settings
        props.put(ConsumerConfig.MAX_POLL_RECORDS_CONFIG, 1000);
        props.put(ConsumerConfig.FETCH_MIN_BYTES_CONFIG, 1024 * 1024); // 1MB
        props.put(ConsumerConfig.FETCH_MAX_WAIT_MS_CONFIG, 1000);

        return new DefaultKafkaConsumerFactory<>(props);
    }

    private ConsumerFactory<String, Object> highThroughputBatchConsumerFactory() {
        Map<String, Object> props = new HashMap<>();
        props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        props.put(ConsumerConfig.GROUP_ID_CONFIG, "high-throughput-batch-group");
        props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class);
        props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
JsonDeserializer.class);

        // Maximum throughput settings
        props.put(ConsumerConfig.MAX_POLL_RECORDS_CONFIG, 2000);
        props.put(ConsumerConfig.FETCH_MIN_BYTES_CONFIG, 2 * 1024 * 1024); // 2MB
        props.put(ConsumerConfig.FETCH_MAX_WAIT_MS_CONFIG, 500);
        props.put(ConsumerConfig.RECEIVE_BUFFER_CONFIG, 128 * 1024); // 128KB
        return new DefaultKafkaConsumerFactory<>(props);
    }

    private ConsumerFactory<String, Object> orderedBatchConsumerFactory() {
        Map<String, Object> props = new HashMap<>();
        props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        props.put(ConsumerConfig.GROUP_ID_CONFIG, "ordered-batch-group");
        props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class);
        props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
JsonDeserializer.class);

        // Ordered processing settings
        props.put(ConsumerConfig.MAX_POLL_RECORDS_CONFIG, 100);
        props.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, false);

        return new DefaultKafkaConsumerFactory<>(props);
    }

    private BatchErrorHandler batchErrorHandler() {
        return new DefaultBatchErrorHandler();
    }
}

/**
 * Advanced batch message listeners
 */
@Component
@lombok.extern.slf4j.Slf4j
public class BatchMessageListeners {

    /**
     * High-throughput batch processing
     */
    @KafkaListener(
        topics = "high-volume-events",
        groupId = "batch-processor",
        containerFactory = "batchContainerFactory"
    )
    public void processBatchEvents(
            List<EventMessage> events,
            List<ConsumerRecord<String, EventMessage>> records,
            Acknowledgment ack) {

        long startTime = System.currentTimeMillis();
        log.info("Processing batch of {} events", events.size());

        try {
            // Process events in parallel within the batch
            events.parallelStream().forEach(this::processEvent);

            // Acknowledge entire batch
            ack.acknowledge();
            long duration = System.currentTimeMillis() - startTime;
            log.info("Successfully processed batch of {} events in {}ms",
                events.size(), duration);

        } catch (Exception e) {
            log.error("Batch processing failed", e);
            // Handle partial batch failure
            handleBatchProcessingError(events, records, e);
            throw e;
        }
    }

    /**
     * Maximum throughput batch processing
     */
    @KafkaListener(
        topics = "ultra-high-volume",
        groupId = "ultra-high-throughput",
        containerFactory = "highThroughputBatchFactory"
    )
    public void processUltraHighThroughputBatch(
            List<Object> messages,
            @Header(KafkaHeaders.RECEIVED_TOPIC) List<String> topics,
            @Header(KafkaHeaders.RECEIVED_PARTITION_ID) List<Integer> partitions,
            Acknowledgment ack) {

        log.info("Processing ultra-high-throughput batch: {} messages across {}
partitions",
            messages.size(), partitions.stream().distinct().count());

        try {
            // Group by message type for efficient processing
            Map<Class<?>, List<Object>> messagesByType = messages.stream()
                .collect(Collectors.groupingBy(Object::getClass));

            // Process each type separately
            messagesByType.forEach(this::processBatchByType);

            ack.acknowledge();

        } catch (Exception e) {
            log.error("Ultra-high-throughput batch processing failed", e);
            throw e;
        }
    }

    /**
     * Ordered batch processing
     */
    @KafkaListener(
        topics = "ordered-transactions",
        groupId = "ordered-batch-processor",
        containerFactory = "orderedBatchFactory"
    )
    public void processOrderedBatch(
            List<TransactionEvent> transactions,
            Acknowledgment ack) {

        log.info("Processing ordered batch of {} transactions",
transactions.size());

        try {
            // Sequential processing to maintain order
            for (TransactionEvent transaction : transactions) {
                processTransactionInOrder(transaction);
            }

            ack.acknowledge();

        } catch (Exception e) {
            log.error("Ordered batch processing failed", e);
            throw e;
        }
    }

    /**
     * Batch processing with partial failure recovery
     */
    @KafkaListener(
        topics = "resilient-batch-events",
        groupId = "resilient-batch-processor",
        containerFactory = "batchContainerFactory"
    )
    public void processResilientBatch(
            List<ResilientEvent> events,
            List<ConsumerRecord<String, ResilientEvent>> records,
            Acknowledgment ack) {

        log.info("Processing resilient batch of {} events", events.size());

        List<ResilientEvent> successful = new ArrayList<>();
        List<ResilientEvent> failed = new ArrayList<>();

        for (int i = 0; i < events.size(); i++) {
            try {
                ResilientEvent event = events.get(i);
                processResilientEvent(event);
                successful.add(event);

            } catch (Exception e) {
                failed.add(events.get(i));
                log.error("Failed to process event at index {}: {}",
                    i, records.get(i).offset(), e);
            }
        }

        log.info("Batch processing completed: {} successful, {} failed",
            successful.size(), failed.size());

        if (failed.isEmpty()) {
            // All successful - acknowledge entire batch
            ack.acknowledge();
        } else {
            // Partial failure - handle failed messages
            handlePartialBatchFailure(failed, records);
            // Still acknowledge batch to avoid reprocessing successful messages
            ack.acknowledge();
        }
    }

    /**
     * Database batch insert optimization
     */
    @KafkaListener(
        topics = "database-inserts",
        groupId = "database-batch-processor",
        containerFactory = "batchContainerFactory"
    )
    @Transactional
    public void processDatabaseBatch(
            List<DatabaseRecord> records,
            Acknowledgment ack) {

        log.info("Processing database batch of {} records", records.size());

        try {
            // Batch database operations for efficiency
            batchInsertToDatabase(records);

            ack.acknowledge();

        } catch (Exception e) {
            log.error("Database batch processing failed", e);
            throw e;
        }
    }

    // Business logic methods
    private void processEvent(EventMessage event) {
        log.debug("Processing event: {}", event.getEventId());
    }

    private void processBatchByType(Class<?> messageType, List<Object> messages) {
        log.debug("Processing batch of {} messages of type: {}",
            messages.size(), messageType.getSimpleName());
    }

    private void processTransactionInOrder(TransactionEvent transaction) {
        log.debug("Processing transaction in order: {}",
transaction.getTransactionId());
    }
```

```
    private void processResilientEvent(ResilientEvent event) {
        log.debug("Processing resilient event: {}", event.getEventId());

        // Simulate occasional failures
        if (event.getEventId().endsWith("FAIL")) {
            throw new RuntimeException("Simulated processing failure");
        }
    }

    private void handleBatchProcessingError(List<EventMessage> events,
                                          List<ConsumerRecord<String,
EventMessage>> records,
                                          Exception e) {
        log.error("Handling batch processing error for {} events", events.size());
        // Could implement individual event retry or DLQ logic
    }

    private void handlePartialBatchFailure(List<ResilientEvent> failed,
                                         List<ConsumerRecord<String,
ResilientEvent>> records) {
        log.warn("Handling partial batch failure for {} events", failed.size());
        // Could send failed events to DLQ or retry topic
    }

    private void batchInsertToDatabase(List<DatabaseRecord> records) {
        log.debug("Performing batch database insert for {} records",
records.size());
        // Implement batch database insert logic
    }
}

// Domain objects
@lombok.Data
@lombok.Builder
@lombok.NoArgsConstructor
@lombok.AllArgsConstructor
class ResilientEvent {
    private String eventId;
    private String eventType;
    private String payload;
    private java.time.Instant timestamp;
}

@lombok.Data
@lombok.Builder
@lombok.NoArgsConstructor
@lombok.AllArgsConstructor
class DatabaseRecord {
    private String recordId;
    private String data;
    private java.time.Instant timestamp;
}
```

Manual vs Auto Ack Modes

Simple Explanation: Acknowledgment modes control when Kafka considers a message as successfully processed. Auto acknowledgment provides convenience but may lead to message loss, while manual acknowledgment offers control but requires careful implementation.

Comprehensive Acknowledgment Mode Configuration

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.kafka.config.ConcurrentKafkaListenerContainerFactory;
import org.springframework.kafka.listener.ContainerProperties;
import org.springframework.kafka.support.Acknowledgment;

/**
 * Advanced acknowledgment mode configurations
 */
@Configuration
@lombok.extern.slf4j.Slf4j
public class AcknowledgmentConfiguration {

    /**
     * Manual immediate acknowledgment container
     * Most control, best for critical processing
     */
    @Bean
    public ConcurrentKafkaListenerContainerFactory<String, Object>
manualImmediateAckFactory() {
        ConcurrentKafkaListenerContainerFactory<String, Object> factory =
            new ConcurrentKafkaListenerContainerFactory<>();

        factory.setConsumerFactory(consumerFactory());

        // Manual immediate acknowledgment

factory.getContainerProperties().setAckMode(ContainerProperties.AckMode.MANUAL_IMM
EDIATE);

        // Optimized for manual ack
        factory.getContainerProperties().setAckTime(5000);
        factory.getContainerProperties().setSyncCommits(false);

        return factory;
    }

    /**
     * Manual acknowledgment container
     * Batched commits for performance
     */
    @Bean
    public ConcurrentKafkaListenerContainerFactory<String, Object>
```

```java
manualAckFactory() {
        ConcurrentKafkaListenerContainerFactory<String, Object> factory =
            new ConcurrentKafkaListenerContainerFactory<>();

        factory.setConsumerFactory(consumerFactory());

        // Manual acknowledgment (batched)

factory.getContainerProperties().setAckMode(ContainerProperties.AckMode.MANUAL);
        factory.getContainerProperties().setAckTime(10000); // 10 second batch
window

        return factory;
    }

    /**
     * Record-level acknowledgment container
     * Commit after each record
     */
    @Bean
    public ConcurrentKafkaListenerContainerFactory<String, Object>
recordAckFactory() {
        ConcurrentKafkaListenerContainerFactory<String, Object> factory =
            new ConcurrentKafkaListenerContainerFactory<>();

        factory.setConsumerFactory(consumerFactory());

        // Record acknowledgment

factory.getContainerProperties().setAckMode(ContainerProperties.AckMode.RECORD);

        return factory;
    }

    /**
     * Batch acknowledgment container
     * Commit after batch processing
     */
    @Bean
    public ConcurrentKafkaListenerContainerFactory<String, Object>
batchAckFactory() {
        ConcurrentKafkaListenerContainerFactory<String, Object> factory =
            new ConcurrentKafkaListenerContainerFactory<>();

        factory.setConsumerFactory(consumerFactory());

        // Batch acknowledgment

factory.getContainerProperties().setAckMode(ContainerProperties.AckMode.BATCH);
        factory.setBatchListener(true);

        return factory;
    }
    /**
     * Time-based acknowledgment container
     * Commit based on time intervals
     */
    @Bean
    public ConcurrentKafkaListenerContainerFactory<String, Object>
timeAckFactory() {
        ConcurrentKafkaListenerContainerFactory<String, Object> factory =
            new ConcurrentKafkaListenerContainerFactory<>();

        factory.setConsumerFactory(consumerFactory());

        // Time-based acknowledgment

factory.getContainerProperties().setAckMode(ContainerProperties.AckMode.TIME);
        factory.getContainerProperties().setAckTime(5000); // 5 seconds

        return factory;
    }

    /**
     * Count-based acknowledgment container
     * Commit after N messages
     */
    @Bean
    public ConcurrentKafkaListenerContainerFactory<String, Object>
countAckFactory() {
        ConcurrentKafkaListenerContainerFactory<String, Object> factory =
            new ConcurrentKafkaListenerContainerFactory<>();

        factory.setConsumerFactory(consumerFactory());

        // Count-based acknowledgment

factory.getContainerProperties().setAckMode(ContainerProperties.AckMode.COUNT);
        factory.getContainerProperties().setAckCount(100); // Every 100 messages

        return factory;
    }

    /**
     * Count and time acknowledgment container
     * Commit based on count OR time, whichever comes first
     */
    @Bean
    public ConcurrentKafkaListenerContainerFactory<String, Object>
countTimeAckFactory() {
        ConcurrentKafkaListenerContainerFactory<String, Object> factory =
            new ConcurrentKafkaListenerContainerFactory<>();

        factory.setConsumerFactory(consumerFactory());

        // Count and time acknowledgment
factory.getContainerProperties().setAckMode(ContainerProperties.AckMode.COUNT_TIME
);
        factory.getContainerProperties().setAckCount(50);   // 50 messages
        factory.getContainerProperties().setAckTime(3000);  // OR 3 seconds

        return factory;
    }

    private ConsumerFactory<String, Object> consumerFactory() {
        // Implementation would be here
        return null;
    }
}

/**
 * Acknowledgment mode examples and patterns
 */
@Component
@lombok.extern.slf4j.Slf4j
public class AcknowledgmentModeListeners {

    /**
     * Manual immediate acknowledgment - most control
     * Acknowledge immediately after successful processing
     */
    @KafkaListener(
        topics = "critical-orders",
        groupId = "critical-order-processor",
        containerFactory = "manualImmediateAckFactory"
    )
    public void processCriticalOrdersManualImmediate(@Payload OrderEvent order,
                                                     Acknowledgment ack) {

        log.info("Processing critical order with manual immediate ack: {}",
order.getOrderId());

        try {
            // Process the order
            processCriticalOrder(order);

            // Acknowledge immediately after successful processing
            ack.acknowledge();

            log.info("Successfully processed and acknowledged critical order: {}",
                order.getOrderId());

        } catch (Exception e) {
            log.error("Failed to process critical order: {}", order.getOrderId(),
e);
            // Don't acknowledge - message will be redelivered
            throw e;
        }
    }
    /**
     * Manual acknowledgment - batched commits
     * Better performance through batching
     */
    @KafkaListener(
        topics = "normal-orders",
        groupId = "normal-order-processor",
        containerFactory = "manualAckFactory"
    )
    public void processNormalOrdersManual(@Payload OrderEvent order,
                                           Acknowledgment ack) {

        log.info("Processing normal order with manual ack: {}",
order.getOrderId());

        try {
            processNormalOrder(order);

            // Acknowledge - will be batched with other acknowledgments
            ack.acknowledge();

        } catch (Exception e) {
            log.error("Failed to process normal order: {}", order.getOrderId(),
e);
            throw e;
        }
    }

    /**
     * Conditional acknowledgment based on processing result
     */
    @KafkaListener(
        topics = "conditional-processing",
        groupId = "conditional-processor",
        containerFactory = "manualImmediateAckFactory"
    )
    public void processConditionalAcknowledgment(@Payload ProcessingTask task,
                                                  Acknowledgment ack) {

        log.info("Processing task with conditional ack: {}", task.getTaskId());

        try {
            ProcessingResult result = processTask(task);

            if (result.isSuccess()) {
                // Acknowledge successful processing
                ack.acknowledge();
                log.info("Task processed successfully and acknowledged: {}",
task.getTaskId());
            } else if (result.isRetryable()) {
                // Don't acknowledge - will be retried
                log.warn("Task processing failed (retryable): {}",
task.getTaskId());
            } else {
                // Acknowledge to avoid infinite retries for non-retryable
failures
                ack.acknowledge();
                log.error("Task processing failed (non-retryable), acknowledged to
avoid retry: {}",
                    task.getTaskId());
                sendToDeadLetterQueue(task, result.getError());
            }

        } catch (Exception e) {
            log.error("Exception processing task: {}", task.getTaskId(), e);
            throw e;
        }
    }

    /**
     * Batch processing with manual acknowledgment
     */
    @KafkaListener(
        topics = "batch-events",
        groupId = "batch-manual-processor",
        containerFactory = "batchAckFactory"
    )
    public void processBatchManualAck(List<EventMessage> events,
                                      Acknowledgment ack) {

        log.info("Processing batch with manual ack: {} events", events.size());

        try {
            // Process all events in batch
            for (EventMessage event : events) {
                processEvent(event);
            }

            // Acknowledge entire batch after successful processing
            ack.acknowledge();

            log.info("Successfully processed and acknowledged batch of {} events",
events.size());

        } catch (Exception e) {
            log.error("Batch processing failed", e);
            // Don't acknowledge - entire batch will be redelivered
            throw e;
        }
    }

    /**
     * Partial batch acknowledgment pattern
     */
    @KafkaListener(
        topics = "partial-batch-events",
        groupId = "partial-batch-processor",
        containerFactory = "manualImmediateAckFactory"
    )
    public void processPartialBatchAck(List<EventMessage> events,
                                       List<ConsumerRecord<String, EventMessage>>
records,
                                       Acknowledgment ack,
                                       Consumer<String, EventMessage> consumer) {

        log.info("Processing partial batch: {} events", events.size());

        int processed = 0;
        List<TopicPartition> partitionsToSeek = new ArrayList<>();

        for (int i = 0; i < events.size(); i++) {
            try {
                EventMessage event = events.get(i);
                ConsumerRecord<String, EventMessage> record = records.get(i);

                processEvent(event);
                processed++;

            } catch (Exception e) {
                log.error("Failed to process event at index {}: offset={}",
                    i, records.get(i).offset(), e);

                // Seek to failed message for retry
                TopicPartition partition = new TopicPartition(
                    records.get(i).topic(),
                    records.get(i).partition()
                );
                partitionsToSeek.add(partition);

                // Stop processing remaining messages in this batch
                break;
            }
        }

        if (partitionsToSeek.isEmpty()) {
            // All events processed successfully
            ack.acknowledge();
            log.info("Successfully processed entire batch: {} events", processed);
        } else {
            // Seek to first failed message
            for (TopicPartition partition : partitionsToSeek) {
                long seekOffset = records.get(processed).offset();
                consumer.seek(partition, seekOffset);
                log.info("Seeking partition {} to offset {} for retry", partition,
seekOffset);
            }

            // Don't acknowledge - will restart from failed message
            log.warn("Partial batch processing: {} successful, seeking for retry",
processed);
        }
    }
    /**
     * Auto acknowledgment example (for comparison)
     */
    @KafkaListener(
        topics = "auto-ack-events",
        groupId = "auto-ack-processor"
        // Uses default container factory with auto acknowledgment
    )
    public void processAutoAck(@Payload EventMessage event) {
        log.info("Processing event with auto ack: {}", event.getEventId());

        try {
            processEvent(event);
            // No manual acknowledgment needed - automatic after method returns

        } catch (Exception e) {
            log.error("Failed to process event: {}", event.getEventId(), e);
            // Message is still acknowledged automatically!
            // This can lead to message loss
            throw e;
        }
    }

    /**
     * Acknowledgment with timeout handling
     */
    @KafkaListener(
        topics = "timeout-sensitive",
        groupId = "timeout-processor",
        containerFactory = "manualImmediateAckFactory"
    )
    public void processWithTimeout(@Payload TimeoutSensitiveMessage message,
                                  Acknowledgment ack) {

        log.info("Processing timeout-sensitive message: {}",
message.getMessageId());

        CompletableFuture<Void> processingFuture = CompletableFuture.runAsync(() -
> {
            try {
                processTimeoutSensitiveMessage(message);
            } catch (Exception e) {
                throw new RuntimeException(e);
            }
        });

        try {
            // Wait for processing with timeout
            processingFuture.get(30, java.util.concurrent.TimeUnit.SECONDS);

            // Acknowledge after successful processing
            ack.acknowledge();
            log.info("Successfully processed timeout-sensitive message: {}",
                message.getMessageId());

        } catch (java.util.concurrent.TimeoutException e) {
            log.error("Processing timeout for message: {}", 
message.getMessageId());
            // Don't acknowledge - will be retried
            processingFuture.cancel(true);
        } catch (Exception e) {
            log.error("Failed to process timeout-sensitive message: {}", 
                message.getMessageId(), e);
            throw new RuntimeException(e);
        }
    }

    // Business logic methods
    private void processCriticalOrder(OrderEvent order) {
        log.debug("Processing critical order: {}", order.getOrderId());
        // Simulate processing time
        try {
            Thread.sleep(100);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }

    private void processNormalOrder(OrderEvent order) {
        log.debug("Processing normal order: {}", order.getOrderId());
    }

    private ProcessingResult processTask(ProcessingTask task) {
        log.debug("Processing task: {}", task.getTaskId());

        // Simulate different processing outcomes
        if (task.getTaskId().endsWith("SUCCESS")) {
            return ProcessingResult.success();
        } else if (task.getTaskId().endsWith("RETRY")) {
            return ProcessingResult.retryableFailure("Temporary failure");
        } else {
            return ProcessingResult.nonRetryableFailure("Permanent failure");
        }
    }

    private void processEvent(EventMessage event) {
        log.debug("Processing event: {}", event.getEventId());

        // Simulate occasional failures
        if (event.getEventId().endsWith("FAIL")) {
            throw new RuntimeException("Simulated processing failure");
        }
    }

    private void sendToDeadLetterQueue(ProcessingTask task, String error) {
        log.warn("Sending task to dead letter queue: taskId={}, error={}", 
            task.getTaskId(), error);
    }

    private void processTimeoutSensitiveMessage(TimeoutSensitiveMessage message) {
        log.debug("Processing timeout-sensitive message: {}",
message.getMessageId());

        // Simulate long processing
        try {
            Thread.sleep(message.getProcessingTimeMs());
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
            throw new RuntimeException("Processing interrupted", e);
        }
    }
}

// Supporting classes
@lombok.Data
@lombok.AllArgsConstructor
@lombok.NoArgsConstructor
class ProcessingResult {
    private boolean success;
    private boolean retryable;
    private String error;

    public static ProcessingResult success() {
        return new ProcessingResult(true, false, null);
    }

    public static ProcessingResult retryableFailure(String error) {
        return new ProcessingResult(false, true, error);
    }

    public static ProcessingResult nonRetryableFailure(String error) {
        return new ProcessingResult(false, false, error);
    }
}

@lombok.Data
@lombok.Builder
@lombok.NoArgsConstructor
@lombok.AllArgsConstructor
class ProcessingTask {
    private String taskId;
    private String taskType;
    private String payload;
    private java.time.Instant timestamp;
}

@lombok.Data
@lombok.Builder
@lombok.NoArgsConstructor
@lombok.AllArgsConstructor
class TimeoutSensitiveMessage {
    private String messageId;
    private long processingTimeMs;
    private String content;
    private java.time.Instant timestamp;
}
```

Acknowledgment Mode Comparison

Mode	When Committed	Performance	Reliability	Use Case
AUTO	After listener returns	Highest	Lowest	Development, non-critical
MANUAL_IMMEDIATE	Immediately on ack()	Medium	Highest	Critical processing
MANUAL	Batched commits	High	High	High-throughput reliable
RECORD	After each record	Low	High	Ordered processing
BATCH	After batch processing	High	Medium	Batch operations
TIME	Every N seconds	Medium	Medium	Time-based commits
COUNT	Every N messages	Medium	Medium	Count-based commits
COUNT_TIME	N messages OR N seconds	Medium	Medium	Hybrid approach

This comprehensive guide covers all aspects of Spring Kafka consumer side implementation with extensive examples and best practices. The document provides both beginner-friendly explanations and advanced patterns for experienced developers, with complete working code examples for production use.