Spring Kafka: Complete Integration Guide

A comprehensive guide covering Spring Kafka integration with the Spring ecosystem, from fundamental concepts to advanced enterprise patterns with extensive Java examples and best practices.

Table of Contents

🌱 Introduction & Basics

What is Spring Kafka?

Simple Explanation: Spring Kafka is a Spring project that provides a Spring-friendly integration layer on top of the native Apache Kafka Java client, offering higher-level abstractions, Spring ecosystem integration, and declarative configuration through annotations.

Problem It Solves:

- **Boilerplate Reduction**: Eliminates repetitive Kafka client setup code
- **Spring Integration**: Seamless integration with Spring's dependency injection, transactions, and configuration management
- **Error Handling**: Sophisticated error handling and retry mechanisms
- **Testing Support**: Built-in testing utilities for Kafka integration tests
- **Monitoring Integration**: Easy integration with Spring Actuator and metrics

Why It Exists in Kafka Ecosystem:

- Raw Kafka clients require significant boilerplate and configuration management
- Spring applications benefit from declarative configuration and dependency injection
- Need for standardized patterns in enterprise Spring applications
- Integration with Spring's transaction management and error handling paradigms

Internal Architecture

```
Spring Kafka Architecture:

┌────────────────────────────────────────────────────────────────┐
│                    Spring Application Layer                    │
├────────────────────────────────────────────────────────────────┤
│                     Spring Kafka Layer                         │
│  ┌────────────────┐   ┌────────────────┐   ┌────────────────┐  │
│  │   KafkaTemplate│   │ @KafkaListener │   │  Configuration │  │
│  │   (Producer)   │   │   (Consumer)   │   │   Management   │  │
│  └────────────────┘   └────────────────┘   └────────────────┘  │
│  ┌────────────────┐   ┌────────────────┐   ┌────────────────┐  │
│  │Message Listener│   │ Error Handlers │   │ Retry Logic    │  │
│  │   Containers   │   │ & Dead Letter  │   │ & Transactions │  │
│  └────────────────┘   └────────────────┘   └────────────────┘  │
├────────────────────────────────────────────────────────────────┤
│                   Native Kafka Client Layer                    │
│  ┌────────────────┐   ┌────────────────┐   ┌────────────────┐  │
│  │  KafkaProducer │   │  KafkaConsumer │   │   AdminClient  │  │
│  └────────────────┘   └────────────────┘   └────────────────┘  │
└────────────────────────────────────────────────────────────────┘
                              ↓
┌────────────────────────────────────────────────────────────────┐
│                     Apache Kafka Cluster                       │
└────────────────────────────────────────────────────────────────┘
```

Integration with Spring Ecosystem

Spring Kafka integrates deeply with core Spring features:

Spring Feature	Spring Kafka Integration
Dependency Injection	Auto-wired producers, consumers, and configurations
Configuration Management	`@ConfigurationProperties` for Kafka settings
Transaction Management	Declarative transactions with `@Transactional`
Error Handling	Global exception handlers and retry mechanisms
Testing	`@EmbeddedKafka` for integration tests
Actuator	Health checks and metrics endpoints
Security	Integration with Spring Security
Cloud Config	External configuration management

Advantages over Plain Kafka Client

Aspect	Plain Kafka Client	Spring Kafka	Winner
Setup Complexity	High boilerplate code	Minimal configuration	Spring Kafka

Aspect	Plain Kafka Client	Spring Kafka	Winner
Error Handling	Manual implementation	Built-in sophisticated handling	Spring Kafka
Testing	Complex test setup	`@EmbeddedKafka` support	Spring Kafka
Configuration	Programmatic configuration	Declarative with properties	Spring Kafka
Monitoring	Manual JMX setup	Automatic Actuator integration	Spring Kafka
Performance	Maximum control	Slight overhead	Plain Client
Flexibility	Complete freedom	Framework conventions	Plain Client
Learning Curve	Kafka + Java knowledge	Spring + Kafka knowledge	Depends

🔧 Key Components

Producer

KafkaTemplate - The Heart of Spring Kafka Producer

Simple Explanation: KafkaTemplate is Spring Kafka's central class for producing messages, providing a high-level abstraction over the native Kafka producer with Spring-style configuration and error handling.

Internal Architecture:

- Wraps the native `KafkaProducer`
- Manages connection pooling and lifecycle
- Provides synchronous and asynchronous sending options
- Integrates with Spring's transaction management

Comprehensive Producer Implementation

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

import java.util.concurrent.CompletableFuture;
import java.time.LocalDateTime;
import java.util.UUID;

/**
 * Comprehensive Spring Kafka Producer implementation
 * Demonstrates various sending patterns, error handling, and Spring integration
 */
@Service
public class SpringKafkaProducerService {

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

    @Autowired
    private KafkaTemplate<String, Object> kafkaTemplate;

    // Topic names - externalize to application.properties
    private static final String ORDERS_TOPIC = "orders";
    private static final String NOTIFICATIONS_TOPIC = "notifications";
    private static final String AUDIT_TOPIC = "audit-events";

    /**
     * Simple synchronous message sending
     */
    public void sendSimpleMessage(String topic, String key, Object message) {
        try {
            // Synchronous send - blocks until completion
            SendResult<String, Object> result = kafkaTemplate.send(topic, key,
message).get();

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

        } catch (Exception e) {
            logger.error("Failed to send message to topic: {}", topic, e);
            throw new RuntimeException("Message sending failed", e);
        }
    }

    /**
     * Asynchronous message sending with callback
     */
    public void sendAsyncMessage(String topic, String key, Object message) {
        CompletableFuture<SendResult<String, Object>> future =
            kafkaTemplate.send(topic, key, message);

        future.whenComplete((result, ex) -> {
            if (ex != null) {
                logger.error("Failed to send message to topic: {} with key: {}",
topic, key, ex);
                handleSendFailure(topic, key, message, ex);
            } else {
                logger.info("Message sent successfully: topic={}, key={},
partition={}, offset={}",
                    topic, key,
                    result.getRecordMetadata().partition(),
                    result.getRecordMetadata().offset());
                handleSendSuccess(result);
            }
        });
    }

    /**
     * Advanced message sending with headers and custom configuration
     */
    public void sendAdvancedMessage(Object payload, String correlationId, String eventType) {
        try {
            Message<Object> message = MessageBuilder
                .withPayload(payload)
                .setHeader(KafkaHeaders.TOPIC, ORDERS_TOPIC)
                .setHeader(KafkaHeaders.KEY, generateMessageKey(payload))
                .setHeader("correlation-id", correlationId)
                .setHeader("event-type", eventType)
                .setHeader("timestamp", System.currentTimeMillis())
                .setHeader("source", "order-service")
                .setHeader("version", "1.0")
                .build();

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

            future.whenComplete((result, ex) -> {
                if (ex != null) {
                    logger.error("Failed to send advanced message with correlationId: {}", correlationId, ex);
                    // Could implement dead letter topic logic here
                    sendToDeadLetterTopic(message, ex);
                } else {
                    logger.info("Advanced message sent: correlationId={}, partition={}, offset={}",
                        correlationId,
                        result.getRecordMetadata().partition(),
                        result.getRecordMetadata().offset());
                }
            });

        } catch (Exception e) {
            logger.error("Error creating advanced message", e);
            throw new RuntimeException("Advanced message creation failed", e);
        }
    }

    /**
     * Transactional message sending
     * Demonstrates Spring's declarative transaction management with Kafka
     */
    @Transactional("kafkaTransactionManager")
    public void sendTransactionalMessages(OrderEvent orderEvent) {
        try {
            // Send order event
            sendMessage(ORDERS_TOPIC, orderEvent.getOrderId(), orderEvent);

            // Send notification event (part of same transaction)
            NotificationEvent notification =
createNotificationFromOrder(orderEvent);
            sendMessage(NOTIFICATIONS_TOPIC, orderEvent.getUserId(),
notification);

            // Send audit event (part of same transaction)
            AuditEvent audit = createAuditEvent(orderEvent);
            sendMessage(AUDIT_TOPIC, orderEvent.getOrderId(), audit);

            logger.info("All transactional messages sent successfully for order:
{}",
                orderEvent.getOrderId());

        } catch (Exception e) {
            logger.error("Transactional message sending failed for order: {}",
                orderEvent.getOrderId(), e);
            // Transaction will be rolled back automatically
            throw e;
        }
    }

    /**
     * Batch message sending for high throughput scenarios
     */
    public void sendBatchMessages(java.util.List<OrderEvent> orders) {
        java.util.List<CompletableFuture<SendResult<String, Object>>> futures =
            new java.util.ArrayList<>();

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

        // Wait for all messages to be sent
        CompletableFuture<Void> allFutures = CompletableFuture.allOf(
            futures.toArray(new CompletableFuture[0]));

        allFutures.whenComplete((result, ex) -> {
            if (ex != null) {
                logger.error("Batch sending failed", ex);
                handleBatchSendFailure(orders, ex);
            } else {
                logger.info("Batch of {} messages sent successfully",
orders.size());

                // Log individual results
                futures.forEach(future -> {
                    try {
                        SendResult<String, Object> sendResult = future.get();
                        logger.debug("Batch item sent: partition={}, offset={}",
                            sendResult.getRecordMetadata().partition(),
                            sendResult.getRecordMetadata().offset());
                    } catch (Exception e) {
                        logger.warn("Error getting batch result", e);
                    }
                });
            }
        });
    }

    /**
     * Send message with custom partitioning
     */
    public void sendWithCustomPartition(String topic, String key, Object message,
int partition) {
        try {
            SendResult<String, Object> result = kafkaTemplate.send(topic,
partition, key, message).get();

            logger.info("Message sent to specific partition: topic={}, partition=
{}, offset={}",
                result.getRecordMetadata().topic(),
                result.getRecordMetadata().partition(),
                result.getRecordMetadata().offset());

        } catch (Exception e) {
            logger.error("Failed to send message to partition {} of topic: {}",
partition, topic, e);
            throw new RuntimeException("Partitioned message sending failed", e);
        }
    }

    /**
     * Request-Reply pattern using ReplyingKafkaTemplate
     */
    public String sendRequestReplyMessage(String request) {
        try {
            // This would require ReplyingKafkaTemplate configuration
            // CompletableFuture<String> replyFuture =
replyingTemplate.sendAndReceive(record);
            // return replyFuture.get(10, TimeUnit.SECONDS);

            // Placeholder implementation
            logger.info("Request-reply message sent: {}", request);
            return "Reply for: " + request;

        } catch (Exception e) {
            logger.error("Request-reply failed for request: {}", request, e);
            throw new RuntimeException("Request-reply failed", e);
        }
    }

    // Helper methods
    private void sendMessage(String topic, String key, Object message) {
        kafkaTemplate.send(topic, key, message);
    }

    private String generateMessageKey(Object payload) {
        if (payload instanceof OrderEvent) {
            return ((OrderEvent) payload).getOrderId();
        }
        return UUID.randomUUID().toString();
    }

    private void handleSendSuccess(SendResult<String, Object> result) {
        // Implement success handling logic
        // Could update metrics, logs, etc.
    }

    private void handleSendFailure(String topic, String key, Object message,
Throwable ex) {
        // Implement failure handling logic
        // Could retry, send to DLQ, alert, etc.
        logger.error("Send failure - Topic: {}, Key: {}, Error: {}", topic, key,
ex.getMessage());
    }

    private void sendToDeadLetterTopic(Message<Object> message, Throwable ex) {
        try {
            String dlqTopic = ORDERS_TOPIC + ".DLT";

            Message<Object> dlqMessage = MessageBuilder
                .fromMessage(message)
                .setHeader(KafkaHeaders.TOPIC, dlqTopic)
                .setHeader("original-topic",
message.getHeaders().get(KafkaHeaders.TOPIC))
                .setHeader("error-message", ex.getMessage())
                .setHeader("error-timestamp", System.currentTimeMillis())
                .build();

            kafkaTemplate.send(dlqMessage);
            logger.info("Message sent to dead letter topic: {}", dlqTopic);

        } catch (Exception dlqEx) {
            logger.error("Failed to send message to dead letter topic", dlqEx);
        }
    }

    private NotificationEvent createNotificationFromOrder(OrderEvent orderEvent) {
        return NotificationEvent.builder()
            .userId(orderEvent.getUserId())
            .message("Order " + orderEvent.getOrderId() + " has been processed")
            .type("ORDER_CONFIRMATION")
            .timestamp(LocalDateTime.now())
            .build();
    }

    private AuditEvent createAuditEvent(OrderEvent orderEvent) {
        return AuditEvent.builder()
            .entityId(orderEvent.getOrderId())
            .entityType("ORDER")
            .action("CREATED")
            .userId(orderEvent.getUserId())
            .timestamp(LocalDateTime.now())
            .build();
    }

    private void handleBatchSendFailure(java.util.List<OrderEvent> orders, Throwable ex) {
        logger.error("Batch send failure for {} orders", orders.size(), ex);
        // Could implement individual retry logic, DLQ, or alerting
    }
}

/**
 * Order event domain object
 */
@lombok.Data
@lombok.Builder
@lombok.NoArgsConstructor
@lombok.AllArgsConstructor
public class OrderEvent {
    private String orderId;
    private String userId;
    private String productId;
    private Integer quantity;
    private java.math.BigDecimal price;
    private String status;
    private LocalDateTime timestamp;
}

/**
 * Notification event domain object
 */
@lombok.Data
@lombok.Builder
@lombok.NoArgsConstructor
@lombok.AllArgsConstructor
public class NotificationEvent {
    private String userId;
    private String message;
    private String type;
    private LocalDateTime timestamp;
}

/**
 * Audit event domain object
 */
@lombok.Data
@lombok.Builder
@lombok.NoArgsConstructor
@lombok.AllArgsConstructor
```

```
public class AuditEvent {
    private String entityId;
    private String entityType;
    private String action;
    private String userId;
    private LocalDateTime timestamp;
}
```

Consumer

Spring Kafka Consumer with @KafkaListener

Simple Explanation: Spring Kafka consumers use the `@KafkaListener` annotation to declaratively consume messages from Kafka topics, providing automatic deserialization, error handling, and integration with Spring's container management.

Internal Architecture:

- Managed by `KafkaListenerEndpointRegistry`
- Uses `MessageListenerContainer` for lifecycle management
- Automatic offset management and consumer group coordination
- Built-in error handling and retry mechanisms

Comprehensive Consumer Implementation

```
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.retry.annotation.Backoff;
import org.springframework.retry.annotation.Retryable;
import org.springframework.stereotype.Component;
import org.springframework.transaction.annotation.Transactional;

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

import java.util.List;
import java.time.LocalDateTime;

/**
 * Comprehensive Spring Kafka Consumer implementation
 * Demonstrates various consumption patterns, error handling, and Spring
integration
 */
@Component
@lombok.extern.slf4j.Slf4j
```

```
public class SpringKafkaConsumerService {

    /**
     * Simple message consumption with automatic acknowledgment
     */
    @KafkaListener(topics = "orders", groupId = "order-processing-group")
    public void consumeOrderEvents(@Payload OrderEvent orderEvent) {
        try {
            log.info("Received order event: orderId={}, status={}",
                orderEvent.getOrderId(), orderEvent.getStatus());

            // Process the order
            processOrder(orderEvent);

            log.info("Successfully processed order: {}", orderEvent.getOrderId());

        } catch (Exception e) {
            log.error("Error processing order event: {}", orderEvent.getOrderId(),
e);
            // Error handling is managed by container error handler
            throw e;
        }
    }

    /**
     * Advanced consumption with headers and manual acknowledgment
     */
    @KafkaListener(
        topics = "notifications",
        groupId = "notification-service-group",
        containerFactory = "kafkaListenerContainerFactory"
    )
    public void consumeNotifications(
            @Payload NotificationEvent notification,
            @Header(KafkaHeaders.RECEIVED_TOPIC) String topic,
            @Header(KafkaHeaders.RECEIVED_PARTITION_ID) int partition,
            @Header(KafkaHeaders.OFFSET) long offset,
            @Header(value = "correlation-id", required = false) String
correlationId,
            ConsumerRecord<String, NotificationEvent> record,
            Acknowledgment acknowledgment) {

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

            // Process notification with full context
            processNotificationWithContext(notification, record);

            // Manual acknowledgment - commit offset only after successful
processing
            acknowledgment.acknowledge();
            log.info("Successfully processed notification for user: {}", notification.getUserId());

        } catch (Exception e) {
            log.error("Error processing notification: partition={}, offset={}", partition, offset, e);
            // Don't acknowledge - message will be redelivered based on retry configuration
            throw e;
        }
    }

    /**
     * Batch message consumption for high throughput
     */
    @KafkaListener(
        topics = "audit-events",
        groupId = "audit-batch-group",
        containerFactory = "batchListenerContainerFactory"
    )
    public void consumeAuditEventsBatch(List<AuditEvent> auditEvents,
                                       List<ConsumerRecord<String, AuditEvent>> records,
                                       Acknowledgment acknowledgment) {
        try {
            log.info("Processing batch of {} audit events", auditEvents.size());

            // Process events in batch for efficiency
            processBatchAuditEvents(auditEvents, records);

            // Acknowledge entire batch
            acknowledgment.acknowledge();

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

        } catch (Exception e) {
            log.error("Error processing audit events batch", e);
            // Implement partial batch recovery if needed
            handleBatchProcessingError(auditEvents, records, e);
            throw e;
        }
    }

    /**
     * Consumption with retry logic and dead letter topic
     */
    @KafkaListener(topics = "payment-events", groupId = "payment-processing-group")
    @Retryable(
        value = {Exception.class},
        maxAttempts = 3,
        backoff = @Backoff(delay = 1000, multiplier = 2)
    )
    public void consumePaymentEvents(@Payload PaymentEvent paymentEvent,
                                   ConsumerRecord<String, PaymentEvent> record) {
        try {
            log.info("Processing payment event: paymentId={}, amount={}",
                paymentEvent.getPaymentId(), paymentEvent.getAmount());

            // Simulate processing that might fail
            processPaymentWithValidation(paymentEvent);

            log.info("Successfully processed payment: {}",
paymentEvent.getPaymentId());

        } catch (Exception e) {
            log.error("Error processing payment event: {}, attempt will be
retried",
                paymentEvent.getPaymentId(), e);
            throw e;
        }
    }

    /**
     * Consumption from specific partitions with offset management
     */
    @KafkaListener(
        topicPartitions = {
            @TopicPartition(topic = "user-events", partitions = {"0", "1"}),
            @TopicPartition(topic = "user-events", partitions = "2",
                partitionOffsets = @PartitionOffset(partition = "2", initialOffset
= "100"))
        },
        groupId = "user-service-specific-partitions"
    )
    public void consumeUserEventsFromSpecificPartitions(
            @Payload UserEvent userEvent,
            @Header(KafkaHeaders.RECEIVED_PARTITION_ID) int partition) {

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

            // Process based on partition for ordered processing
            processUserEventByPartition(userEvent, partition);

        } catch (Exception e) {
            log.error("Error processing user event from partition {}: {}",
                partition, userEvent.getUserId(), e);
            throw e;
        }
    }

    /**
     * Transactional message consumption
     */
    @KafkaListener(topics = "inventory-events", groupId = "inventory-transaction-
group")
    @Transactional("transactionManager")
    public void consumeInventoryEventsTransactionally(@Payload InventoryEvent
inventoryEvent) {
        try {
            log.info("Processing inventory event transactionally: productId={},
quantity={}",
                inventoryEvent.getProductId(), inventoryEvent.getQuantity());

            // Database operations will be part of the transaction
            updateInventoryInDatabase(inventoryEvent);

            // Send downstream events (will be part of same transaction if using
transactional producer)
            publishInventoryUpdatedEvent(inventoryEvent);

            log.info("Successfully processed inventory event: {}",
inventoryEvent.getProductId());

        } catch (Exception e) {
            log.error("Error in transactional processing of inventory event: {}",
                inventoryEvent.getProductId(), e);
            // Transaction will be rolled back
            throw e;
        }
    }

    /**
     * Message filtering with SpEL expressions
     */
    @KafkaListener(
        topics = "all-events",
        groupId = "filtered-events-group",
        filter = "filterBean"
    )
    public void consumeFilteredEvents(@Payload Object event) {
        log.info("Processing filtered event: {}", event);
        // Only events that pass the filter will reach here
    }

    /**
     * Consumption with custom deserializer handling
     */
    @KafkaListener(
        topics = "json-events",
        groupId = "json-processing-group",
        properties = {
            "spring.json.trusted.packages=com.example.events",

"spring.json.type.mapping=order:com.example.events.OrderEvent,notification:com.exa
mple.events.NotificationEvent"
        }
    )
    public void consumeJsonEvents(@Payload Object event,
                                @Header(KafkaHeaders.RECEIVED_TOPIC) String topic)
{
        try {
            log.info("Processing JSON event from topic {}: {}", topic,
event.getClass().getSimpleName());

            // Handle different event types
            if (event instanceof OrderEvent) {
                processOrder((OrderEvent) event);
            } else if (event instanceof NotificationEvent) {
                processNotification((NotificationEvent) event);
            } else {
                log.warn("Unknown event type: {}",
event.getClass().getSimpleName());
            }

        } catch (Exception e) {
            log.error("Error processing JSON event from topic {}", topic, e);
            throw e;
        }
    }

    /**
     * Error handling method for failed messages
     */
    @KafkaListener(
        topics = "orders.DLT", // Dead Letter Topic
        groupId = "dlt-processing-group"
    )
    public void handleDeadLetterMessages(
            @Payload Object failedMessage,
            @Header(KafkaHeaders.RECEIVED_TOPIC) String topic,
            @Header(value = "kafka_dlt-original-topic", required = false) String
originalTopic,
            @Header(value = "kafka_dlt-exception-message", required = false)
String errorMessage) {

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

        try {
            // Implement dead letter processing logic
            processDeadLetterMessage(failedMessage, originalTopic, errorMessage);

        } catch (Exception e) {
            log.error("Failed to process dead letter message", e);
            // Could send to another DLT or alert administrators
        }
    }

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

        // Could involve database updates, external API calls, etc.
        if ("INVALID".equals(orderEvent.getStatus())) {
            throw new RuntimeException("Invalid order status");
        }
    }

    private void processNotificationWithContext(NotificationEvent notification,
                                              ConsumerRecord<String,
NotificationEvent> record) {
        // Process notification with full Kafka record context
        log.debug("Processing notification with full context: userId={},
timestamp={}",
            notification.getUserId(), record.timestamp());
    }

    private void processBatchAuditEvents(List<AuditEvent> events,
                                       List<ConsumerRecord<String, AuditEvent>>
records) {
        // Batch processing for efficiency
        for (int i = 0; i < events.size(); i++) {
            AuditEvent event = events.get(i);
            ConsumerRecord<String, AuditEvent> record = records.get(i);

            log.debug("Processing audit event {}/{}: entityId={}",
                i + 1, events.size(), event.getEntityId());
        }
    }

    private void processPaymentWithValidation(PaymentEvent paymentEvent) {
        // Simulate payment processing with potential failures
        if (paymentEvent.getAmount().compareTo(java.math.BigDecimal.ZERO) <= 0) {
            throw new IllegalArgumentException("Invalid payment amount");
        }

        // Simulate external payment processor call
        if (paymentEvent.getPaymentId().endsWith("FAIL")) {
            throw new RuntimeException("Payment processor unavailable");
        }
    }

    private void processUserEventByPartition(UserEvent userEvent, int partition) {
        // Partition-specific processing logic
        log.debug("Partition-specific processing for user {} on partition {}",
            userEvent.getUserId(), partition);
    }

    private void updateInventoryInDatabase(InventoryEvent inventoryEvent) {
        // Database update logic (will be transactional)
        log.debug("Updating inventory in database for product: {}",
            inventoryEvent.getProductId());
    }

    private void publishInventoryUpdatedEvent(InventoryEvent inventoryEvent) {
        // Publish downstream event (part of same transaction if configured)
        log.debug("Publishing inventory updated event for product: {}",
            inventoryEvent.getProductId());
    }

    private void handleBatchProcessingError(List<AuditEvent> auditEvents,
                                          List<ConsumerRecord<String, AuditEvent>>
records,
                                          Exception e) {
        // Implement partial batch recovery logic
        log.warn("Implementing batch error recovery for {} events",
auditEvents.size());
    }

    private void processDeadLetterMessage(Object failedMessage, String
originalTopic, String errorMessage) {
        // Implement dead letter processing
        log.debug("Processing dead letter message from {}: {}", originalTopic,
errorMessage);
    }

    private void processNotification(NotificationEvent notification) {
        log.debug("Processing notification for user: {}",
notification.getUserId());
    }
}

// Additional domain objects
@lombok.Data
@lombok.Builder
@lombok.NoArgsConstructor
@lombok.AllArgsConstructor
class PaymentEvent {
    private String paymentId;
    private String orderId;
    private java.math.BigDecimal amount;
    private String currency;
    private String status;
    private LocalDateTime timestamp;
}

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

```
@lombok.Data
@lombok.Builder
@lombok.NoArgsConstructor
@lombok.AllArgsConstructor
class InventoryEvent {
    private String productId;
    private Integer quantity;
    private String operation; // ADD, REMOVE, SET
    private LocalDateTime timestamp;
}
```

Message Listener Containers

Simple Explanation: Message Listener Containers are the runtime components that manage the lifecycle of Kafka consumers, handling connection management, partition assignment, offset commits, and error recovery.

Internal Architecture:

```
Message Listener Container Architecture:

┌─────────────────────────────────────────────────────────────────┐
│           ConcurrentMessageListenerContainer                    │
├─────────────────────────────────────────────────────────────────┤
│  ┌─────────────────┐  ┌─────────────────┐  ┌─────────────────┐ │
│  │KafkaMessage     │  │KafkaMessage     │  │KafkaMessage     │ │
│  │ListenerContainer│  │ListenerContainer│  │ListenerContainer│ │
│  │   (Thread 1)    │  │   (Thread 2)    │  │   (Thread 3)    │ │
│  └─────────────────┘  └─────────────────┘  └─────────────────┘ │
├─────────────────────────────────────────────────────────────────┤
│                    Container Properties                         │
│  ┌─────────────────┐  ┌─────────────────┐  ┌─────────────────┐ │
│  │   Poll Config   │  │ Error Handlers  │  │Commit Strategies│ │
│  └─────────────────┘  └─────────────────┘  └─────────────────┘ │
└─────────────────────────────────────────────────────────────────┘
                              ↓
┌─────────────────────────────────────────────────────────────────┐
│                  Native Kafka Consumer                          │
│                     (per thread)                                │
└─────────────────────────────────────────────────────────────────┘
```

Comprehensive Container Configuration

```
import org.springframework.beans.factory.annotation.Value;
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.core.DefaultKafkaConsumerFactory;
import org.springframework.kafka.listener.*;
import org.springframework.kafka.support.Acknowledgment;
import org.springframework.retry.policy.SimpleRetryPolicy;
import org.springframework.retry.support.RetryTemplate;
import org.springframework.util.backoff.FixedBackOff;

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;

/**
 * Comprehensive Message Listener Container configuration
 * Demonstrates various container patterns and configurations
 */
@Configuration
@lombok.extern.slf4j.Slf4j
public class KafkaListenerContainerConfiguration {

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

    @Value("${spring.kafka.consumer.group-id:default-group}")
    private String defaultGroupId;

    /**
     * Default Consumer Factory
     */
    @Bean
    public ConsumerFactory<String, Object> consumerFactory() {
        Map<String, Object> configProps = new HashMap<>();

        // Basic configuration
        configProps.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG,
bootstrapServers);
        configProps.put(ConsumerConfig.GROUP_ID_CONFIG, defaultGroupId);

        // Serialization
        configProps.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class);
        configProps.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
JsonDeserializer.class);

        // Performance tuning
        configProps.put(ConsumerConfig.MAX_POLL_RECORDS_CONFIG, 500);
        configProps.put(ConsumerConfig.MAX_POLL_INTERVAL_MS_CONFIG, 300000);
        configProps.put(ConsumerConfig.SESSION_TIMEOUT_MS_CONFIG, 30000);
        configProps.put(ConsumerConfig.HEARTBEAT_INTERVAL_MS_CONFIG, 10000);

        // Offset management
        configProps.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest");
        configProps.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, false); // Manual commit

        // JSON deserialization
        configProps.put(JsonDeserializer.TRUSTED_PACKAGES, "com.example.events");
        configProps.put(JsonDeserializer.TYPE_MAPPINGS,
            "order:com.example.events.OrderEvent,notification:com.example.events.NotificationEvent");

        return new DefaultKafkaConsumerFactory<>(configProps);
    }

    /**
     * Default Kafka Listener Container Factory
     */
    @Bean
    public ConcurrentKafkaListenerContainerFactory<String, Object> kafkaListenerContainerFactory() {
        ConcurrentKafkaListenerContainerFactory<String, Object> factory =
            new ConcurrentKafkaListenerContainerFactory<>();

        factory.setConsumerFactory(consumerFactory());

        // Concurrency configuration
        factory.setConcurrency(3); // 3 consumer threads

        // Acknowledgment configuration
        factory.getContainerProperties().setAckMode(ContainerProperties.AckMode.MANUAL_IMMEDIATE);

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

        // Record interceptor for logging/monitoring
        factory.setRecordInterceptors(recordInterceptor());

        // Message filtering
        factory.setRecordFilterStrategy(record -> {
            // Example: Filter out test messages
            return record.value() != null &&
                   record.value().toString().contains("TEST");
        });

        return factory;
    }

    /**
     * Batch Listener Container Factory
     */
    @Bean
    public ConcurrentKafkaListenerContainerFactory<String, Object> batchListenerContainerFactory() {
        ConcurrentKafkaListenerContainerFactory<String, Object> factory =
            new ConcurrentKafkaListenerContainerFactory<>();

        factory.setConsumerFactory(consumerFactory());

        // Enable batch processing
        factory.setBatchListener(true);
        factory.setConcurrency(2);

        // Batch configuration

factory.getContainerProperties().setAckMode(ContainerProperties.AckMode.BATCH);
        factory.getContainerProperties().setPollTimeout(3000);

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

        return factory;
    }

    /**
     * High-throughput container factory with optimized settings
     */
    @Bean
    public ConcurrentKafkaListenerContainerFactory<String, Object>
highThroughputContainerFactory() {
        ConcurrentKafkaListenerContainerFactory<String, Object> factory =
            new ConcurrentKafkaListenerContainerFactory<>();

        // Custom consumer factory with optimized settings
        Map<String, Object> props = new HashMap<>();
        props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, bootstrapServers);
        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

        ConsumerFactory<String, Object> optimizedConsumerFactory =
            new DefaultKafkaConsumerFactory<>(props);
        factory.setConsumerFactory(optimizedConsumerFactory);

        // High concurrency
        factory.setConcurrency(8);

        // Async acknowledgment for better performance

factory.getContainerProperties().setAckMode(ContainerProperties.AckMode.MANUAL);
```

```
        return factory;
    }

    /**
     * Manual acknowledgment container with custom commit strategy
     */
    @Bean
    public ConcurrentKafkaListenerContainerFactory<String, Object> 
manualAckContainerFactory() {
        ConcurrentKafkaListenerContainerFactory<String, Object> factory = 
            new ConcurrentKafkaListenerContainerFactory<>();

        factory.setConsumerFactory(consumerFactory());
        factory.setConcurrency(2);

        // Manual acknowledgment configuration
        
factory.getContainerProperties().setAckMode(ContainerProperties.AckMode.MANUAL_IMM
EDIATE);

        // Custom acknowledgment timeout
        factory.getContainerProperties().setAckTime(5000);

        // Custom commit callback
        factory.getContainerProperties().setCommitCallback((offsets, exception) -> 
{
            if (exception != null) {
                log.error("Commit failed for offsets: {}", offsets, exception);
            } else {
                log.debug("Successfully committed offsets: {}", offsets);
            }
        });

        return factory;
    }

    /**
     * Container with custom lifecycle management
     */
    @Bean
    public MessageListenerContainer customLifecycleContainer() {
        ContainerProperties containerProperties = new ContainerProperties("custom-
lifecycle-topic");
        containerProperties.setMessageListener(new MessageListener<String, Object>
() {
            @Override
            public void onMessage(ConsumerRecord<String, Object> record) {
                log.info("Custom container received: {}", record.value());
            }
        });

        // Custom lifecycle configuration
        containerProperties.setConsumerStartTimeout(Duration.ofSeconds(30));
        containerProperties.setShutdownTimeout(Duration.ofSeconds(30));

        ConcurrentMessageListenerContainer<String, Object> container =
            new ConcurrentMessageListenerContainer<>(consumerFactory(),
containerProperties);

        container.setConcurrency(2);
        container.setAutoStartup(true);

        return container;
    }

    /**
     * Default error handler with retry and dead letter topic
     */
    @Bean
    public DefaultErrorHandler defaultErrorHandler() {
        // Retry configuration
        FixedBackOff backOff = new FixedBackOff(1000L, 3); // 1 second delay, 3
retries

        DefaultErrorHandler errorHandler = new DefaultErrorHandler(
            deadLetterPublishingRecoverer(), backOff);

        // Configure which exceptions should be retried
        errorHandler.addNotRetryableExceptions(IllegalArgumentException.class);
        errorHandler.addRetryableExceptions(RuntimeException.class);

        return errorHandler;
    }

    /**
     * Dead letter topic publishing recoverer
     */
    @Bean
    public DeadLetterPublishingRecoverer deadLetterPublishingRecoverer() {
        return new DeadLetterPublishingRecoverer(kafkaTemplate(),
            (record, ex) -> {
                // Custom DLT topic naming strategy
                String originalTopic = record.topic();
                return new org.apache.kafka.common.TopicPartition(originalTopic +
".DLT", -1);
            });
    }

    /**
     * Batch error handler
     */
    @Bean
    public BatchErrorHandler batchErrorHandler() {
        return new DefaultBatchErrorHandler();
    }

    /**
```

```
     * Record interceptor for monitoring and logging
     */
    @Bean
    public RecordInterceptor<String, Object> recordInterceptor() {
        return new RecordInterceptor<String, Object>() {
            @Override
            public ConsumerRecord<String, Object> intercept(ConsumerRecord<String, Object> record) {
                log.debug("Intercepting record: topic={}, partition={}, offset={}",
                    record.topic(), record.partition(), record.offset());

                // Add monitoring/metrics logic here
                recordProcessingMetrics(record);

                return record;
            }

            @Override
            public void success(ConsumerRecord<String, Object> record, Object listener) {
                log.debug("Successfully processed record: topic={}, offset={}",
                    record.topic(), record.offset());
            }

            @Override
            public void failure(ConsumerRecord<String, Object> record, Exception exception, Object listener) {
                log.error("Failed to process record: topic={}, offset={}",
                    record.topic(), record.offset(), exception);
            }
        };
    }

    /**
     * Container registry for programmatic container management
     */
    @Bean
    public KafkaListenerEndpointRegistry kafkaListenerEndpointRegistry() {
        return new KafkaListenerEndpointRegistry();
    }

    /**
     * Message filtering bean
     */
    @Bean("filterBean")
    public RecordFilterStrategy<String, Object> filterStrategy() {
        return record -> {
            // Example: Filter messages older than 1 hour
            long oneHourAgo = System.currentTimeMillis() - (60 * 60 * 1000);
            return record.timestamp() < oneHourAgo;
        };
    }
    // Helper methods
    private void recordProcessingMetrics(ConsumerRecord<String, Object> record) {
        // Implement metrics collection
        // Could use Micrometer metrics here
    }

    // Placeholder for KafkaTemplate bean (would be defined in producer configuration)
    private org.springframework.kafka.core.KafkaTemplate<String, Object> kafkaTemplate() {
        // This would typically be injected
        return null;
    }
}
```

KafkaTemplate

Simple Explanation: KafkaTemplate is Spring Kafka's central template class for sending messages to Kafka topics, providing a simplified API over the native Kafka producer with Spring's conventions for configuration, error handling, and integration.

Key Features:

- Synchronous and asynchronous message sending
- Spring-style configuration and dependency injection
- Built-in error handling and retry mechanisms
- Transaction support with Spring's transaction management
- Metrics and monitoring integration

Advanced KafkaTemplate Configuration

```java
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.kafka.core.*;
import org.springframework.kafka.transaction.KafkaTransactionManager;
import org.springframework.kafka.support.ProducerListener;
import org.springframework.kafka.support.serializer.JsonSerializer;

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

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

/**
 * Comprehensive KafkaTemplate configuration and customization
 */
```

```
@Configuration
@lombok.extern.slf4j.Slf4j
public class KafkaTemplateConfiguration {

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

    /**
     * Default Producer Factory
     */
    @Bean
    public ProducerFactory<String, Object> producerFactory() {
        Map<String, Object> configProps = new HashMap<>();

        // Basic configuration
        configProps.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,
bootstrapServers);

        // Serialization
        configProps.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
StringSerializer.class);
        configProps.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
JsonSerializer.class);

        // Reliability settings
        configProps.put(ProducerConfig.ACKS_CONFIG, "all");
        configProps.put(ProducerConfig.RETRIES_CONFIG, Integer.MAX_VALUE);
        configProps.put(ProducerConfig.ENABLE_IDEMPOTENCE_CONFIG, true);

        // Performance tuning
        configProps.put(ProducerConfig.BATCH_SIZE_CONFIG, 16384);
        configProps.put(ProducerConfig.LINGER_MS_CONFIG, 5);
        configProps.put(ProducerConfig.BUFFER_MEMORY_CONFIG, 33554432);
        configProps.put(ProducerConfig.COMPRESSION_TYPE_CONFIG, "lz4");

        // Timeout settings
        configProps.put(ProducerConfig.REQUEST_TIMEOUT_MS_CONFIG, 30000);
        configProps.put(ProducerConfig.DELIVERY_TIMEOUT_MS_CONFIG, 120000);

        return new DefaultKafkaProducerFactory<>(configProps);
    }

    /**
     * Default KafkaTemplate
     */
    @Bean
    public KafkaTemplate<String, Object> kafkaTemplate() {
        KafkaTemplate<String, Object> template = new KafkaTemplate<>
(producerFactory());

        // Set default topic
        template.setDefaultTopic("default-topic");

        // Add producer listener for monitoring
        template.setProducerListener(defaultProducerListener());

        // Set message converter if needed
        // template.setMessageConverter(new StringJsonMessageConverter());

        return template;
    }

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

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

        // Transaction configuration
        configProps.put(ProducerConfig.TRANSACTIONAL_ID_CONFIG, "tx-producer-1");
        configProps.put(ProducerConfig.ENABLE_IDEMPOTENCE_CONFIG, true);
        configProps.put(ProducerConfig.ACKS_CONFIG, "all");

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

        return factory;
    }

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

        return template;
    }

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

    /**
     * High-throughput KafkaTemplate for batch operations
     */
    @Bean
    public KafkaTemplate<String, Object> highThroughputKafkaTemplate() {
        Map<String, Object> configProps = new HashMap<>();

        configProps.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, bootstrapServers);
        configProps.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, StringSerializer.class);
        configProps.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, JsonSerializer.class);

        // High-throughput optimizations
        configProps.put(ProducerConfig.BATCH_SIZE_CONFIG, 65536); // 64KB
        configProps.put(ProducerConfig.LINGER_MS_CONFIG, 10); // Higher linger time
        configProps.put(ProducerConfig.BUFFER_MEMORY_CONFIG, 67108864); // 64MB
        configProps.put(ProducerConfig.COMPRESSION_TYPE_CONFIG, "lz4");

        // Trade some durability for performance
        configProps.put(ProducerConfig.ACKS_CONFIG, "1");
        configProps.put(ProducerConfig.RETRIES_CONFIG, 3);

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

        template.setProducerListener(performanceProducerListener());

        return template;
    }

    /**
     * Request-Reply KafkaTemplate (ReplyingKafkaTemplate)
     */
    @Bean
    public ReplyingKafkaTemplate<String, Object, Object> replyingKafkaTemplate() {
        // This requires a consumer container for replies
        ConcurrentMessageListenerContainer<String, Object> replyContainer =
            replyContainer();

        ReplyingKafkaTemplate<String, Object, Object> template =
            new ReplyingKafkaTemplate<>(producerFactory(), replyContainer);

        template.setDefaultReplyTimeout(Duration.ofSeconds(10));
        template.setSharedReplyTopic(true);

        return template;
    }

    /**
     * Reply container for request-reply pattern
     */
    private ConcurrentMessageListenerContainer<String, Object> replyContainer() {
        ContainerProperties containerProperties = new ContainerProperties("reply-
topic");
        containerProperties.setGroupId("reply-group-" +
java.util.UUID.randomUUID());

        ConcurrentMessageListenerContainer<String, Object> container =
            new ConcurrentMessageListenerContainer<>(consumerFactory(),
containerProperties);
        container.setConcurrency(1);
        container.setAutoStartup(false); // Will be started by
ReplyingKafkaTemplate

        return container;
    }

    /**
     * Default producer listener for monitoring
     */
    @Bean
    public ProducerListener<String, Object> defaultProducerListener() {
        return new ProducerListener<String, Object>() {
            @Override
            public void onSuccess(ProducerRecord<String, Object> producerRecord,
                                RecordMetadata recordMetadata) {
                log.debug("Message sent successfully: topic={}, partition={},
offset={}",
                    recordMetadata.topic(), recordMetadata.partition(),
recordMetadata.offset());

                // Add success metrics
                recordSuccessMetrics(producerRecord, recordMetadata);
            }

            @Override
            public void onError(ProducerRecord<String, Object> producerRecord,
                              RecordMetadata recordMetadata, Exception exception)
{
                log.error("Message send failed: topic={}, key={}",
                    producerRecord.topic(), producerRecord.key(), exception);

                // Add error metrics and alerting
                recordErrorMetrics(producerRecord, exception);
            }
        };
    }

    /**
     * Transactional producer listener
     */
    @Bean
    public ProducerListener<String, Object> transactionalProducerListener() {
        return new ProducerListener<String, Object>() {
            @Override
            public void onSuccess(ProducerRecord<String, Object> producerRecord,
                                RecordMetadata recordMetadata) {
```

```
                log.debug("Transactional message sent: topic={}, partition={}, 
offset={}",
                    recordMetadata.topic(), recordMetadata.partition(), 
recordMetadata.offset());
            }

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

                // Handle transaction failures
                handleTransactionFailure(producerRecord, exception);
            }
        };
    }

    /**
     * Performance-focused producer listener
     */
    @Bean
    public ProducerListener<String, Object> performanceProducerListener() {
        return new ProducerListener<String, Object>() {
            @Override
            public void onSuccess(ProducerRecord<String, Object> producerRecord,
                                RecordMetadata recordMetadata) {
                // Minimal logging for performance
                if (log.isTraceEnabled()) {
                    log.trace("High-throughput message sent: {}-{}-{}",
                        recordMetadata.topic(), recordMetadata.partition(), 
recordMetadata.offset());
                }
            }

            @Override
            public void onError(ProducerRecord<String, Object> producerRecord,
                              RecordMetadata recordMetadata, Exception exception)
{
                // Only log errors for performance template
                log.warn("High-throughput message failed: {}",
                    producerRecord.topic(), exception);
            }
        };
    }

    /**
     * Custom KafkaTemplate with routing logic
     */
    @Bean
    public KafkaTemplate<String, Object> routingKafkaTemplate() {
        KafkaTemplate<String, Object> template = new KafkaTemplate<String, Object>
(producerFactory()) {
```

```
            @Override
            protected org.springframework.messaging.Message<?>
doSend(ProducerRecord<String, Object> producerRecord) {
                // Custom routing logic based on message content
                String routedTopic =
determineTopicByContent(producerRecord.value());

                ProducerRecord<String, Object> routedRecord = new ProducerRecord<>
(
                    routedTopic,
                    producerRecord.partition(),
                    producerRecord.timestamp(),
                    producerRecord.key(),
                    producerRecord.value(),
                    producerRecord.headers()
                );

                return super.doSend(routedRecord);
            }
        };

        return template;
    }

    // Helper methods
    private void recordSuccessMetrics(ProducerRecord<String, Object> record,
RecordMetadata metadata) {
        // Implement success metrics
        // Could use Micrometer here
    }

    private void recordErrorMetrics(ProducerRecord<String, Object> record,
Exception exception) {
        // Implement error metrics and alerting
    }

    private void handleTransactionFailure(ProducerRecord<String, Object> record,
Exception exception) {
        // Implement transaction failure handling
    }

    private String determineTopicByContent(Object content) {
        // Implement content-based routing
        if (content instanceof OrderEvent) {
            return "orders";
        } else if (content instanceof NotificationEvent) {
            return "notifications";
        }
        return "default-topic";
    }

    // Consumer factory for reply container (placeholder)
    private ConsumerFactory<String, Object> consumerFactory() {
        Map<String, Object> props = new HashMap<>();
```

```
props.put(org.apache.kafka.clients.consumer.ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, bootstrapServers);

props.put(org.apache.kafka.clients.consumer.ConsumerConfig.GROUP_ID_CONFIG, "reply-group");

props.put(org.apache.kafka.clients.consumer.ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, StringSerializer.class);

props.put(org.apache.kafka.clients.consumer.ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG, JsonSerializer.class);

        return new DefaultKafkaConsumerFactory<>(props);
    }
}
```

⚙️ Configuration & Setup

Complete Spring Boot Configuration

Application Properties Configuration

```
# application.properties - Comprehensive Spring Kafka Configuration

# Basic Kafka Configuration
spring.kafka.bootstrap-servers=localhost:9092
spring.kafka.client-id=spring-kafka-app

# Producer Configuration
spring.kafka.producer.key-serializer=org.apache.kafka.common.serialization.StringSerializer
spring.kafka.producer.value-serializer=org.springframework.kafka.support.serializer.JsonSerializer
spring.kafka.producer.acks=all
spring.kafka.producer.retries=2147483647
spring.kafka.producer.enable-idempotence=true
spring.kafka.producer.batch-size=16384
spring.kafka.producer.buffer-memory=33554432
spring.kafka.producer.compression-type=lz4
spring.kafka.producer.linger-ms=5
spring.kafka.producer.request-timeout-ms=30000
spring.kafka.producer.delivery-timeout-ms=120000

# Consumer Configuration
spring.kafka.consumer.key-deserializer=org.apache.kafka.common.serialization.StringDeserializer
spring.kafka.consumer.value-deserializer=org.springframework.kafka.support.serializer.JsonDeserializer
spring.kafka.consumer.group-id=spring-kafka-group
spring.kafka.consumer.auto-offset-reset=earliest
spring.kafka.consumer.enable-auto-commit=false
spring.kafka.consumer.max-poll-records=500
spring.kafka.consumer.max-poll-interval-ms=300000
spring.kafka.consumer.session-timeout-ms=30000
spring.kafka.consumer.heartbeat-interval-ms=10000
spring.kafka.consumer.fetch-min-size=1024
spring.kafka.consumer.fetch-max-wait=500ms

# JSON Serialization Configuration
spring.kafka.producer.properties.spring.json.type.mapping=order:com.example.events.OrderEvent,notification:com.example.events.NotificationEvent
spring.kafka.consumer.properties.spring.json.trusted.packages=com.example.events
spring.kafka.consumer.properties.spring.json.type.mapping=order:com.example.events.OrderEvent,notification:com.example.events.NotificationEvent

# Listener Container Configuration
spring.kafka.listener.ack-mode=manual_immediate
spring.kafka.listener.concurrency=3
spring.kafka.listener.poll-timeout=3000ms
spring.kafka.listener.type=batch
spring.kafka.listener.missing-topics-fatal=false

# Transaction Configuration
spring.kafka.producer.transaction-id-prefix=tx-
spring.kafka.transaction.commit-timeout=10s

# Security Configuration (if needed)
spring.kafka.security.protocol=SASL_SSL
spring.kafka.ssl.trust-store-location=classpath:kafka.client.truststore.jks
spring.kafka.ssl.trust-store-password=password
spring.kafka.ssl.key-store-location=classpath:kafka.client.keystore.jks
spring.kafka.ssl.key-store-password=password
spring.kafka.sasl.mechanism=SCRAM-SHA-512
spring.kafka.sasl.jaas.config=org.apache.kafka.common.security.scram.ScramLoginModule required username="kafka-user" password="kafka-password";

# Admin Configuration
spring.kafka.admin.fail-fast=true
spring.kafka.admin.properties.request.timeout.ms=60000

# Health Check Configuration
management.health.kafka.enabled=true

# Metrics Configuration
management.endpoints.web.exposure.include=health,info,metrics,prometheus
management.endpoint.health.show-details=always
management.metrics.export.prometheus.enabled=true

# Logging Configuration
logging.level.org.springframework.kafka=INFO
logging.level.org.apache.kafka=WARN
logging.level.com.example=DEBUG
```

Complete Java Configuration Class

```
import org.springframework.boot.autoconfigure.kafka.KafkaProperties;
import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Primary;
import org.springframework.kafka.annotation.EnableKafka;
import org.springframework.kafka.annotation.EnableKafkaRetry;
import org.springframework.kafka.config.TopicBuilder;
import org.springframework.kafka.core.KafkaAdmin;
import org.springframework.kafka.listener.ContainerProperties;
import org.springframework.kafka.support.converter.RecordMessageConverter;
import org.springframework.kafka.support.converter.StringJsonMessageConverter;
import org.springframework.retry.annotation.EnableRetry;

import java.util.Map;

/**
 * Complete Spring Kafka Configuration
 */
@Configuration
@EnableKafka
@EnableRetry
@EnableKafkaRetry
@lombok.extern.slf4j.Slf4j
public class SpringKafkaConfiguration {

    /**
     * Kafka Admin for topic management
     */
    @Bean
    public KafkaAdmin kafkaAdmin(KafkaProperties properties) {
        Map<String, Object> configs = properties.buildAdminProperties();

        KafkaAdmin admin = new KafkaAdmin(configs);
        admin.setFatalIfBrokerNotAvailable(true);
        admin.setAutoCreate(true);

        return admin;
    }

    /**
     * Topic definitions
     */
    @Bean
    public org.apache.kafka.clients.admin.NewTopic ordersTopic() {
        return TopicBuilder.name("orders")
            .partitions(6)
            .replicas(3)
```

```
.config(org.apache.kafka.common.config.TopicConfig.RETENTION_MS_CONFIG,
"604800000") // 7 days

.config(org.apache.kafka.common.config.TopicConfig.COMPRESSION_TYPE_CONFIG, "lz4")

.config(org.apache.kafka.common.config.TopicConfig.MIN_IN_SYNC_REPLICAS_CONFIG,
"2")
            .build();
    }

    @Bean
    public org.apache.kafka.clients.admin.NewTopic notificationsTopic() {
        return TopicBuilder.name("notifications")
            .partitions(3)
            .replicas(3)

.config(org.apache.kafka.common.config.TopicConfig.RETENTION_MS_CONFIG,
"259200000") // 3 days
            .build();
    }

    @Bean
    public org.apache.kafka.clients.admin.NewTopic auditTopic() {
        return TopicBuilder.name("audit-events")
            .partitions(1) // Single partition for ordered audit trail
            .replicas(3)

.config(org.apache.kafka.common.config.TopicConfig.RETENTION_MS_CONFIG,
"31536000000") // 1 year

.config(org.apache.kafka.common.config.TopicConfig.CLEANUP_POLICY_CONFIG,
"compact,delete")
            .build();
    }

    @Bean
    public org.apache.kafka.clients.admin.NewTopic deadLetterTopic() {
        return TopicBuilder.name("orders.DLT")
            .partitions(1)
            .replicas(3)

.config(org.apache.kafka.common.config.TopicConfig.RETENTION_MS_CONFIG,
"2592000000") // 30 days
            .build();
    }

    /**
     * Message converter for JSON handling
     */
    @Bean
    public RecordMessageConverter messageConverter() {
        StringJsonMessageConverter converter = new StringJsonMessageConverter();
        converter.setTypeIdPropertyName("__type");
        return converter;
```

```java
    }

    /**
     * Global container properties
     */
    @Bean
    @ConfigurationProperties(prefix = "spring.kafka.listener")
    public ContainerProperties containerProperties() {
        ContainerProperties properties = new ContainerProperties("default");

        // Override with custom settings
        properties.setAckMode(ContainerProperties.AckMode.MANUAL_IMMEDIATE);
        properties.setSyncCommits(false);
        properties.setCommitLogLevel(org.apache.kafka.clients.consumer.internals.AbstractCoordinator.class);

        return properties;
    }

    /**
     * Custom configuration properties
     */
    @Bean
    @ConfigurationProperties(prefix = "app.kafka")
    public CustomKafkaProperties customKafkaProperties() {
        return new CustomKafkaProperties();
    }
}

/**
 * Custom Kafka configuration properties
 */
@lombok.Data
@ConfigurationProperties(prefix = "app.kafka")
public class CustomKafkaProperties {
    private String applicationId = "spring-kafka-app";
    private int retryAttempts = 3;
    private long retryBackoffMs = 1000;
    private boolean enableMetrics = true;
    private boolean enableDeadLetterTopic = true;
    private Map<String, String> topicDefaults = new HashMap<>();
}
```

📊 Comparisons & Trade-offs

Spring Kafka vs Plain Kafka Client

Feature	Spring Kafka	Plain Kafka Client	Winner
Development Speed	Very Fast (annotations, auto-config)	Slow (boilerplate code)	Spring Kafka

Feature	Spring Kafka	Plain Kafka Client	Winner
Learning Curve	Moderate (Spring + Kafka)	High (Kafka internals)	Spring Kafka
Performance	Good (slight overhead)	Excellent (no overhead)	Plain Client
Error Handling	Sophisticated built-in	Manual implementation	Spring Kafka
Testing	Excellent (`@EmbeddedKafka`)	Complex setup	Spring Kafka
Configuration	Declarative (properties/annotations)	Programmatic	Spring Kafka
Monitoring	Built-in Actuator integration	Manual JMX setup	Spring Kafka
Transaction Support	Declarative (`@Transactional`)	Manual management	Spring Kafka
Flexibility	Framework conventions	Complete control	Plain Client
Memory Usage	Higher (Spring context)	Lower (lightweight)	Plain Client
Deployment Size	Larger (Spring dependencies)	Smaller	Plain Client

Message Processing Patterns Comparison

Pattern	Use Case	Advantages	Disadvantages
Single Message Processing	Low-volume, real-time processing	Simple, immediate processing	Lower throughput
Batch Processing	High-volume, latency-tolerant	Higher throughput, efficient	Higher latency, complex error handling
Manual Acknowledgment	Critical reliability requirements	Full control over commits	More complex implementation
Auto Acknowledgment	High-throughput scenarios	Simpler implementation	Risk of message loss

Container Factory Configuration Trade-offs

Configuration	Performance	Reliability	Complexity
High Concurrency	⭐⭐⭐⭐⭐	⭐⭐⭐	⭐⭐⭐⭐
Low Concurrency	⭐⭐	⭐⭐⭐⭐⭐	⭐⭐
Batch Processing	⭐⭐⭐⭐	⭐⭐⭐	⭐⭐⭐⭐
Single Message	⭐⭐⭐	⭐⭐⭐⭐	⭐⭐
Manual Ack	⭐⭐⭐	⭐⭐⭐⭐⭐	⭐⭐⭐⭐
Auto Ack	⭐⭐⭐⭐⭐	⭐⭐	⭐⭐

🚨 Common Pitfalls & Best Practices

Common Pitfalls

❌ Configuration Anti-Patterns

```java
// DON'T - Blocking operations in listener
@KafkaListener(topics = "orders")
public void handleOrder(OrderEvent order) {
    // BLOCKING call - will slow down entire consumer group
    String result = restTemplate.getForObject("http://slow-service/validate", String.class);
    processOrder(order, result);
}

// DON'T - Not handling deserialization errors
@KafkaListener(topics = "events")
public void handleEvent(Object event) {
    // No error handling for malformed JSON
    SomeEvent typedEvent = (SomeEvent) event; // ClassCastException risk
}

// DON'T - Synchronous processing without proper timeout
public void sendMessage(String topic, Object message) {
    try {
        // Blocks indefinitely if broker is down
        SendResult result = kafkaTemplate.send(topic, message).get();
    } catch (Exception e) {
        // Error handling
    }
}
```

❌ Resource Management Issues

```java
// DON'T - Creating too many consumer threads
@Bean
public ConcurrentKafkaListenerContainerFactory<String, Object> containerFactory() {
    factory.setConcurrency(50); // TOO MANY threads for most use cases
    return factory;
}

// DON'T - Not configuring proper timeouts
@Bean
public ConsumerFactory<String, Object> consumerFactory() {
    props.put(ConsumerConfig.MAX_POLL_INTERVAL_MS_CONFIG, 300000); // 5 minutes - too high
    props.put(ConsumerConfig.SESSION_TIMEOUT_MS_CONFIG, 60000); // 1 minute - too high
}
```

Best Practices

✅ Optimal Patterns

```java
/**
 * Best practices implementation
 */
@Component
@lombok.extern.slf4j.Slf4j
public class KafkaBestPracticesExample {

    @Autowired
    private KafkaTemplate<String, Object> kafkaTemplate;

    @Autowired
    private OrderService orderService;

    /**
     * ✅ GOOD - Asynchronous processing with proper error handling
     */
    @KafkaListener(topics = "orders", groupId = "order-processing")
    public void handleOrderAsync(OrderEvent order, Acknowledgment ack) {
        CompletableFuture.supplyAsync(() -> {
            try {
                // Process order asynchronously
                return orderService.processOrder(order);
            } catch (Exception e) {
                log.error("Failed to process order: {}", order.getOrderId(), e);
                throw e;
            }
        }).whenComplete((result, ex) -> {
            if (ex == null) {
                log.info("Successfully processed order: {}", order.getOrderId());
                ack.acknowledge(); // Acknowledge only on success
            } else {
                log.error("Order processing failed: {}", order.getOrderId(), ex);
                // Don't acknowledge - will be retried
            }
        });
    }

    /**
     * ✅ GOOD - Proper error handling with deserialization
     */
    @KafkaListener(topics = "events")
    public void handleEventSafely(@Payload(required = false) Object event,
                                 @Header(KafkaHeaders.RECEIVED_TOPIC) String
topic,
                                 ConsumerRecord<String, Object> record) {
        try {
            if (event == null) {
                log.warn("Received null payload from topic: {}", topic);
                return;
            }

            // Type-safe processing
            if (event instanceof OrderEvent orderEvent) {
                processOrderEvent(orderEvent);
            } else if (event instanceof NotificationEvent notificationEvent) {
                processNotificationEvent(notificationEvent);
            } else {
                log.warn("Unknown event type: {} from topic: {}",
                    event.getClass().getSimpleName(), topic);
            }

        } catch (Exception e) {
            log.error("Error processing event from topic: {}, offset: {}",
                topic, record.offset(), e);
            throw e;
        }
    }

    /**
     * ✅ GOOD - Non-blocking message sending with timeout
     */
    public void sendMessageSafely(String topic, Object message) {
        CompletableFuture<SendResult<String, Object>> future =
            kafkaTemplate.send(topic, message);

        future.orTimeout(10, TimeUnit.SECONDS)
            .whenComplete((result, ex) -> {
                if (ex != null) {
                    if (ex instanceof TimeoutException) {
                        log.error("Message send timeout for topic: {}", topic);
                    } else {
                        log.error("Failed to send message to topic: {}", topic,
ex);
                    }
                    // Implement retry or DLQ logic
                    handleSendFailure(topic, message, ex);
                } else {
                    log.info("Message sent successfully: topic={}, partition={},
offset={}",
                        result.getRecordMetadata().topic(),
                        result.getRecordMetadata().partition(),
                        result.getRecordMetadata().offset());
                }
            });
    }

    /**
     * ✅ GOOD - Batch processing with partial failure handling
     */
    @KafkaListener(topics = "batch-events", containerFactory = "batchContainer")
    public void processBatch(List<EventMessage> events,
                           List<ConsumerRecord<String, EventMessage>> records,
                           Acknowledgment ack) {

        List<EventMessage> successful = new ArrayList<>();
        List<EventMessage> failed = 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);
                successful.add(event);

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

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

        // Handle partial failures
        if (!failed.isEmpty()) {
            handlePartialBatchFailure(failed, records);
        }

        // Acknowledge batch (consider if you want to ack partial failures)
        ack.acknowledge();
    }

    /**
     * ✅ GOOD - Graceful shutdown handling
     */
    @EventListener
    public void handleShutdown(ContextClosedEvent event) {
        log.info("Application shutdown initiated - stopping Kafka containers gracefully");

        // Allow containers to finish processing current messages
        try {
            Thread.sleep(5000); // Give 5 seconds for graceful shutdown
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }

    /**
     * ✅ GOOD - Health check implementation
     */
    @Component
    public static class KafkaHealthIndicator implements HealthIndicator {
        @Autowired
        private KafkaTemplate<String, Object> kafkaTemplate;

        @Override
        public Health health() {
            try {
                // Try to get metadata to check connectivity
                kafkaTemplate.partitionsFor("health-check-topic");
                return Health.up()
                    .withDetail("kafka", "Available")
                    .withDetail("timestamp", Instant.now())
                    .build();

            } catch (Exception e) {
                return Health.down()
                    .withDetail("kafka", "Unavailable")
                    .withDetail("error", e.getMessage())
                    .withDetail("timestamp", Instant.now())
                    .build();
            }
        }
    }

    // Helper methods
    private void processOrderEvent(OrderEvent event) {
        log.debug("Processing order event: {}", event.getOrderId());
    }

    private void processNotificationEvent(NotificationEvent event) {
        log.debug("Processing notification event: {}", event.getUserId());
    }

    private void handleSendFailure(String topic, Object message, Throwable ex) {
        // Implement retry logic or send to DLQ
        log.warn("Implementing send failure handling for topic: {}", topic);
    }

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

    private void handlePartialBatchFailure(List<EventMessage> failed,
                                         List<ConsumerRecord<String,
EventMessage>> records) {
        // Handle failed messages in batch - could send to DLQ
        log.warn("Handling {} failed messages from batch", failed.size());
    }
}

// Domain objects
@lombok.Data
class EventMessage {
    private String id;
    private String type;
    private String data;
}
```

✅ Configuration Best Practices

```
/**
 * Production-ready configuration
 */
@Configuration
@lombok.extern.slf4j.Slf4j
public class ProductionKafkaConfiguration {

    /**
     * ✅ GOOD - Properly tuned consumer factory
     */
    @Bean
    public ConsumerFactory<String, Object> consumerFactory() {
        Map<String, Object> props = new HashMap<>();

        // Connection settings
        props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "${spring.kafka.bootstrap-servers}");

        // Performance tuning
        props.put(ConsumerConfig.MAX_POLL_RECORDS_CONFIG, 500); // Reasonable batch size
        props.put(ConsumerConfig.MAX_POLL_INTERVAL_MS_CONFIG, 120000); // 2 minutes
        props.put(ConsumerConfig.SESSION_TIMEOUT_MS_CONFIG, 30000); // 30 seconds
        props.put(ConsumerConfig.HEARTBEAT_INTERVAL_MS_CONFIG, 10000); // 10 seconds

        // Reliability settings
        props.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, false); // Manual commit
        props.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest");

        // Fetch settings for throughput
        props.put(ConsumerConfig.FETCH_MIN_BYTES_CONFIG, 1024); // 1KB
        props.put(ConsumerConfig.FETCH_MAX_WAIT_MS_CONFIG, 500); // 500ms

        return new DefaultKafkaConsumerFactory<>(props);
    }

    /**
     * ✅ GOOD - Right-sized container factory
     */
    @Bean
    public ConcurrentKafkaListenerContainerFactory<String, Object> containerFactory() {
```

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

        factory.setConsumerFactory(consumerFactory());

        // Right-size concurrency based on partition count and load
        factory.setConcurrency(3); // Good balance for most use cases

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

        // Acknowledgment
        factory.getContainerProperties().setAckMode(ContainerProperties.AckMode.MANUAL_IMMEDIATE);

        return factory;
    }

    /**
     * ✅ GOOD - Comprehensive error handling
     */
    @Bean
    public DefaultErrorHandler productionErrorHandler() {
        // Exponential backoff retry
        ExponentialBackOff backOff = new ExponentialBackOff(1000L, 2.0);
        backOff.setMaxElapsedTime(30000L); // Max 30 seconds of retries

        DefaultErrorHandler errorHandler = new DefaultErrorHandler(
            deadLetterPublishingRecoverer(), backOff);

        // Define non-retryable exceptions
        errorHandler.addNotRetryableExceptions(
            IllegalArgumentException.class,
            DeserializationException.class
        );

        // Add retry listeners for monitoring
        errorHandler.setRetryListeners(retryListener());

        return errorHandler;
    }

    @Bean
    public DeadLetterPublishingRecoverer deadLetterPublishingRecoverer() {
        return new DeadLetterPublishingRecoverer(
            kafkaTemplate(),
            (record, ex) -> new TopicPartition(record.topic() + ".DLT", -1)
        );
    }

    @Bean
    public RetryListener retryListener() {
        return new RetryListener() {
            @Override
            public void onRetry(ConsumerRecord<?, ?> record, Exception ex, int deliveryAttempt) {
                log.warn("Retrying message: topic={}, offset={}, attempt={}",
                    record.topic(), record.offset(), deliveryAttempt);
            }
        };
    }

    // Placeholder for dependencies
    private KafkaTemplate<String, Object> kafkaTemplate() {
        return new KafkaTemplate<>(producerFactory());
    }

    private ProducerFactory<String, Object> producerFactory() {
        // Implementation would be here
        return null;
    }
}
```

🌍 Real-World Use Cases

E-commerce Order Processing System

```
/**
 * Complete e-commerce order processing with Spring Kafka
 */
@Service
@lombok.extern.slf4j.Slf4j
public class EcommerceOrderProcessingService {

    @Autowired
    private KafkaTemplate<String, Object> kafkaTemplate;

    @Autowired
    private OrderService orderService;

    @Autowired
    private InventoryService inventoryService;

    @Autowired
    private PaymentService paymentService;

    /**
     * Order creation event handler
     */
    @KafkaListener(topics = "order-created", groupId = "order-processing")
    @Transactional
    public void handleOrderCreated(OrderCreatedEvent event) {
        log.info("Processing new order: {}", event.getOrderId());
        try {
            // 1. Validate order
            orderService.validateOrder(event);

            // 2. Reserve inventory
            inventoryService.reserveInventory(event.getOrderId(),
event.getItems());

            // 3. Send payment processing event
            PaymentRequestEvent paymentRequest = PaymentRequestEvent.builder()
                .orderId(event.getOrderId())
                .amount(event.getTotalAmount())
                .customerId(event.getCustomerId())
                .build();

            kafkaTemplate.send("payment-requests", event.getOrderId(),
paymentRequest);

            log.info("Order processing initiated: {}", event.getOrderId());

        } catch (Exception e) {
            log.error("Order processing failed: {}", event.getOrderId(), e);

            // Send failure event
            OrderProcessingFailedEvent failureEvent =
OrderProcessingFailedEvent.builder()
                .orderId(event.getOrderId())
                .reason(e.getMessage())
                .originalEvent(event)
                .build();

            kafkaTemplate.send("order-processing-failed", event.getOrderId(),
failureEvent);
            throw e;
        }
    }

    /**
     * Payment confirmation handler
     */
    @KafkaListener(topics = "payment-completed", groupId = "order-fulfillment")
    public void handlePaymentCompleted(PaymentCompletedEvent event) {
        log.info("Payment completed for order: {}", event.getOrderId());

        try {
            // Update order status
            orderService.markOrderAsPaid(event.getOrderId());

            // Trigger fulfillment
            OrderFulfillmentEvent fulfillmentEvent =
OrderFulfillmentEvent.builder()
                .orderId(event.getOrderId())
                .customerId(event.getCustomerId())
                .shippingAddress(event.getShippingAddress())
                .build();

            kafkaTemplate.send("order-fulfillment", event.getOrderId(),
fulfillmentEvent);

            // Send customer notification
            CustomerNotificationEvent notification =
CustomerNotificationEvent.builder()
                .customerId(event.getCustomerId())
                .type("ORDER_CONFIRMED")
                .message("Your order " + event.getOrderId() + " has been confirmed
and will be shipped soon.")
                .build();

            kafkaTemplate.send("customer-notifications", event.getCustomerId(),
notification);

        } catch (Exception e) {
            log.error("Post-payment processing failed for order: {}",
event.getOrderId(), e);
            throw e;
        }
    }
}
```

Microservices Communication Hub

```
/**
 * Microservices communication using Spring Kafka
 */
@Component
@lombok.extern.slf4j.Slf4j
public class MicroservicesCommunicationHub {

    @Autowired
    private KafkaTemplate<String, Object> kafkaTemplate;

    /**
     * User service events
     */
    @KafkaListener(topics = "user-events", groupId = "notification-service")
    public void handleUserEvents(UserEvent event) {
        log.info("Processing user event: {} for user: {}", event.getEventType(),
event.getUserId());

        switch (event.getEventType()) {
            case "USER_REGISTERED" -> handleUserRegistration(event);
            case "USER_UPDATED" -> handleUserUpdate(event);
            case "USER_DELETED" -> handleUserDeletion(event);
            default -> log.warn("Unknown user event type: {}",
event.getEventType());
        }
    }

    /**
     * Order service communication
     */
    @KafkaListener(topics = "inventory-requests", groupId = "inventory-service")
    public void handleInventoryRequests(InventoryRequestEvent event) {
        log.info("Processing inventory request for order: {}",
event.getOrderId());

        try {
            // Process inventory request
            boolean available = checkInventoryAvailability(event);

            InventoryResponseEvent response = InventoryResponseEvent.builder()
                .orderId(event.getOrderId())
                .available(available)
                .reservedItems(available ? event.getRequestedItems() :
Collections.emptyList())
                .build();

            kafkaTemplate.send("inventory-responses", event.getOrderId(),
response);

        } catch (Exception e) {
            log.error("Inventory request processing failed: {}",
event.getOrderId(), e);

            InventoryResponseEvent errorResponse =
InventoryResponseEvent.builder()
                .orderId(event.getOrderId())
                .available(false)
                .error(e.getMessage())
                .build();

            kafkaTemplate.send("inventory-responses", event.getOrderId(),
errorResponse);
        }
    }

    private void handleUserRegistration(UserEvent event) {
        // Send welcome email event
        EmailEvent welcomeEmail = EmailEvent.builder()
            .recipient(event.getUserEmail())
            .template("WELCOME_EMAIL")
            .variables(Map.of("userName", event.getUserName()))
            .build();

        kafkaTemplate.send("email-events", event.getUserId(), welcomeEmail);

        // Create user preferences
        UserPreferencesEvent preferences = UserPreferencesEvent.builder()
```

```
            .userId(event.getUserId())
            .defaultPreferences(getDefaultPreferences())
            .build();

        kafkaTemplate.send("user-preferences", event.getUserId(), preferences);
    }

    private void handleUserUpdate(UserEvent event) {
        log.info("User updated: {}", event.getUserId());
        // Propagate updates to dependent services
    }

    private void handleUserDeletion(UserEvent event) {
        log.info("User deleted: {}", event.getUserId());

        // Trigger cleanup across services
        UserCleanupEvent cleanup = UserCleanupEvent.builder()
            .userId(event.getUserId())
            .requestedBy("user-service")
            .build();

        kafkaTemplate.send("user-cleanup", event.getUserId(), cleanup);
    }

    private boolean checkInventoryAvailability(InventoryRequestEvent event) {
        // Implement inventory check logic
        return true; // Simplified
    }

    private Map<String, Object> getDefaultPreferences() {
        return Map.of(
            "emailNotifications", true,
            "pushNotifications", false,
            "theme", "light"
        );
    }
}
```

Real-time Analytics and Monitoring

```
/**
 * Real-time analytics processing with Spring Kafka
 */
@Component
@lombok.extern.slf4j.Slf4j
public class RealTimeAnalyticsProcessor {

    @Autowired
    private KafkaTemplate<String, Object> kafkaTemplate;

    @Autowired
    private MetricsService metricsService;

    /**
     * Process user activity events for real-time analytics
     */
    @KafkaListener(topics = "user-activity", containerFactory = "batchContainer")
    public void processUserActivityBatch(List<UserActivityEvent> activities,
                                       Acknowledgment ack) {

        log.info("Processing batch of {} user activity events",
activities.size());

        try {
            // Aggregate metrics
            Map<String, Integer> pageViews = new HashMap<>();
            Map<String, Integer> userSessions = new HashMap<>();
            Set<String> activeUsers = new HashSet<>();

            for (UserActivityEvent activity : activities) {
                // Count page views
                pageViews.merge(activity.getPage(), 1, Integer::sum);

                // Count user sessions
                userSessions.merge(activity.getSessionId(), 1, Integer::sum);

                // Track active users
                activeUsers.add(activity.getUserId());
            }

            // Send aggregated metrics
            RealTimeMetricsEvent metrics = RealTimeMetricsEvent.builder()
                .timestamp(Instant.now())
                .pageViews(pageViews)
                .activeSessions(userSessions.size())
                .activeUsers(activeUsers.size())
                .totalEvents(activities.size())
                .build();

            kafkaTemplate.send("realtime-metrics", "metrics", metrics);

            // Update metrics service
            metricsService.updateRealTimeMetrics(metrics);

            ack.acknowledge();

        } catch (Exception e) {
            log.error("Failed to process user activity batch", e);
            throw e;
        }
    }

    /**
     * Process system metrics for monitoring
     */
    @KafkaListener(topics = "system-metrics", groupId = "monitoring-service")
    public void processSystemMetrics(SystemMetricsEvent event) {
        log.debug("Processing system metrics from: {}", event.getSource());

        // Check for alerts
        if (event.getCpuUsage() > 80) {
            AlertEvent alert = AlertEvent.builder()
                .type("HIGH_CPU")
                .severity("WARNING")
                .source(event.getSource())
                .message("High CPU usage: " + event.getCpuUsage() + "%")
                .timestamp(Instant.now())
                .build();

            kafkaTemplate.send("alerts", event.getSource(), alert);
        }

        if (event.getMemoryUsage() > 90) {
            AlertEvent alert = AlertEvent.builder()
                .type("HIGH_MEMORY")
                .severity("CRITICAL")
                .source(event.getSource())
                .message("High memory usage: " + event.getMemoryUsage() + "%")
                .timestamp(Instant.now())
                .build();

            kafkaTemplate.send("alerts", event.getSource(), alert);
        }

        // Store metrics for historical analysis
        metricsService.storeSystemMetrics(event);
    }
}
```

📈 Version Highlights

Spring Kafka Evolution Timeline

Version	Spring Boot	Kafka Client	Key Features
3.1.x	3.2.x	3.6.x	**Enhanced retry mechanisms**, improved error handling
3.0.x	3.0.x	3.4.x	**Spring Boot 3 support**, native compilation support
2.9.x	2.7.x	3.2.x	**Improved batch processing**, enhanced metrics
2.8.x	2.6.x	3.1.x	**ReplyingKafkaTemplate enhancements**, better transaction support
2.7.x	2.5.x	2.8.x	**@RetryableTopic**, enhanced error handling

Version	Spring Boot	Kafka Client	Key Features
2.6.x	2.4.x	2.7.x	**Non-blocking retries**, improved dead letter topics
2.5.x	2.3.x	2.5.x	**Enhanced security**, OAuth support
2.4.x	2.2.x	2.4.x	**Batch listeners**, improved performance
2.3.x	2.1.x	2.3.x	**Request-reply pattern**, ReplyingKafkaTemplate
2.2.x	2.0.x	2.0.x	**Spring Boot 2 support**, reactive support

Modern Features (2023-2025)

Spring Kafka 3.1+ Features:

- **Enhanced Retry Mechanisms**: More sophisticated retry patterns with exponential backoff
- **Better Error Handling**: Improved dead letter topic handling and error recovery
- **Performance Improvements**: Better batch processing and memory management
- **Security Enhancements**: Improved OAuth and SASL support

Spring Boot 3.x Integration:

- **Native Compilation**: GraalVM native image support
- **Improved Autoconfiguration**: Better defaults and configuration properties
- **Observability**: Enhanced metrics and tracing integration
- **Performance**: Better startup time and memory usage

Key Milestones

2.7.x - Game Changer:

- Introduction of `@RetryableTopic` annotation
- Non-blocking retries with automatic dead letter topic creation
- Enhanced batch processing capabilities

2.8.x - Production Ready:

- Improved ReplyingKafkaTemplate for request-reply patterns
- Better transaction support and error handling
- Enhanced monitoring and metrics

3.0.x - Modern Spring:

- Spring Boot 3.0 compatibility
- Native compilation support for faster startup
- Improved configuration and autoconfiguration

🔗 Additional Resources

📚 Official Documentation

- Spring Kafka Reference
- Spring Boot Kafka Properties
- Apache Kafka Documentation

🎓 Learning Resources

- Spring Kafka Samples
- Confluent Spring Boot Tutorial
- Baeldung Spring Kafka Guide

🛠 Testing and Development Tools

- Testcontainers Kafka
- Spring Kafka Test
- EmbeddedKafka

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

> 💡 **Pro Tip**: Spring Kafka excels in Spring ecosystem applications where rapid development, robust error handling, and declarative configuration are priorities. For maximum performance and complete control over Kafka interactions, consider plain Kafka clients, but for most enterprise applications, Spring Kafka provides the perfect balance of functionality and ease of use.