Spring Kafka Batch Processing: Complete Developer Guide

A comprehensive guide covering all aspects of Spring Kafka batch processing, from basic batch listeners to advanced error handling and real-world use cases with extensive Java examples and production patterns.

Table of Contents

What is Batch Processing in Kafka?

Simple Explanation: Batch processing in Kafka allows consuming and processing multiple messages together in a single operation, rather than handling them one by one. This approach significantly improves throughput and reduces processing overhead for high-volume scenarios.

Why Batch Processing Exists:

- **Higher Throughput**: Process thousands of messages in single batches
- **Reduced Overhead**: Fewer method calls and acknowledgments
- **Efficient Resource Usage**: Better CPU and memory utilization
- **Atomic Operations**: Process related messages together
- **Better Latency**: Reduce per-message processing latency

Batch vs Record Processing Architecture:

```
Record-Level Processing:
┌─────────────────────────────────────────────────────────┐
│                    Kafka Topic                          │
│  [Msg1][Msg2][Msg3][Msg4][Msg5][Msg6][Msg7][Msg8]     │
└─────────────────────────────────────────────────────────┘
                         │
                         │ Individual polling
                         ▼
┌─────────────────────────────────────────────────────────┐
│                 Record Listener                         │
│  processMessage(msg1) → processMessage(msg2) → ...     │
│  8 separate method calls                                │
└─────────────────────────────────────────────────────────┘
```

```
Batch Processing:
┌─────────────────────────────────────────────────────────────┐
│                    Kafka Topic                              │
│  [Msg1][Msg2][Msg3][Msg4][Msg5][Msg6][Msg7][Msg8]        │
└─────────────────────────────────────────────────────────────┘
                          │
                          │ Batch polling
                          ▼
┌─────────────────────────────────────────────────────────────┐
│                  Batch Listener                             │
│  processMessages([msg1, msg2, msg3, msg4, msg5, msg6,     │
│                   msg7, msg8])                              │
│  1 method call for 8 messages                              │
└─────────────────────────────────────────────────────────────┘
```

Internal Batch Processing Flow:

```
Kafka Consumer Batch Processing Architecture:

┌─────────────────────────────────────────────────────────────┐
│                  Kafka Brokers                             │
│  Topic: logs    Partition 0: [M1][M2][M3][M4]             │
│                 Partition 1: [M5][M6][M7][M8]             │
│                 Partition 2: [M9][M10][M11][M12]          │
└─────────────────────────────────────────────────────────────┘
                      │
                      │ poll(Duration.ofMillis(100))
                      ▼
┌─────────────────────────────────────────────────────────────┐
│               Consumer Poll Operation                       │
│  ConsumerRecords<K,V> records = consumer.poll(timeout)     │
│  ┌─────────────┐ ┌─────────────┐ ┌─────────────┐          │
│  │ Partition 0 │ │ Partition 1 │ │ Partition 2 │          │
│  │ [M1][M2]    │ │ [M5][M6]    │ │ [M9][M10]   │          │
│  └─────────────┘ └─────────────┘ └─────────────┘          │
└─────────────────────────────────────────────────────────────┘
                      │
                      │ Batch size: 6 messages
                      ▼
┌─────────────────────────────────────────────────────────────┐
│            Spring Kafka Batch Listener                     │
│  @KafkaListener(batch = "true")                            │
│  public void process(List<LogEvent> logs) {                │
│    // Process all 6 messages together                      │
│    batchProcessor.processLogs(logs);                       │
│  }                                                          │
└─────────────────────────────────────────────────────────────┘
```

🔄 Batch Listeners

@KafkaListener with Batch Mode

Basic Batch Listener Configuration

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.kafka.annotation.EnableKafka;
import org.springframework.kafka.annotation.KafkaListener;
import org.springframework.kafka.config.ConcurrentKafkaListenerContainerFactory;
import org.springframework.kafka.core.ConsumerFactory;
import org.springframework.kafka.core.DefaultKafkaConsumerFactory;
import org.springframework.kafka.support.KafkaHeaders;

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

import org.springframework.messaging.handler.annotation.Header;
import org.springframework.messaging.handler.annotation.Payload;

/**
 * Comprehensive batch listener configuration and examples
 */
@Configuration
@EnableKafka
@lombok.extern.slf4j.Slf4j
public class BatchListenerConfiguration {

    /**
     * Basic batch consumer factory configuration
     */
    @Bean
    public ConsumerFactory<String, String> batchConsumerFactory() {
        Map<String, Object> props = new HashMap<>();

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

        // Batch processing configuration
        props.put(ConsumerConfig.MAX_POLL_RECORDS_CONFIG, 500); // Max records per
batch
        props.put(ConsumerConfig.FETCH_MIN_BYTES_CONFIG, 1024 * 50); // 50KB
minimum
        props.put(ConsumerConfig.FETCH_MAX_WAIT_MS_CONFIG, 500); // Max wait for
batch

        // Optimization for batch processing
        props.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest");
        props.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, false); // Manual
commit for batches

        return new DefaultKafkaConsumerFactory<>(props);
    }

    /**
     * Basic batch listener container factory
     */
    @Bean
    public ConcurrentKafkaListenerContainerFactory<String, String>
batchKafkaListenerContainerFactory() {
        ConcurrentKafkaListenerContainerFactory<String, String> factory =
            new ConcurrentKafkaListenerContainerFactory<>();

        factory.setConsumerFactory(batchConsumerFactory());

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

        // Configure acknowledgment mode for batch processing

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

        // Set concurrency for parallel batch processing
        factory.setConcurrency(3);

        log.info("Configured basic batch listener container factory:
maxRecords=500, concurrency=3");

        return factory;
    }

    /**
     * High-throughput batch listener factory for log processing
     */
    @Bean("highThroughputBatchFactory")
    public ConcurrentKafkaListenerContainerFactory<String, String>
highThroughputBatchFactory() {
        Map<String, Object> props = new HashMap<>();

        // High-throughput consumer configuration
        props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        props.put(ConsumerConfig.GROUP_ID_CONFIG, "high-throughput-batch-group");
        props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class);
        props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class);

        // Optimized for high-volume batch processing
        props.put(ConsumerConfig.MAX_POLL_RECORDS_CONFIG, 2000); // Large batches
        props.put(ConsumerConfig.FETCH_MIN_BYTES_CONFIG, 1024 * 1024); // 1MB
minimum
        props.put(ConsumerConfig.FETCH_MAX_WAIT_MS_CONFIG, 100); // Low latency
        props.put(ConsumerConfig.MAX_PARTITION_FETCH_BYTES_CONFIG, 1024 * 1024 *
5); // 5MB per partition

        // Connection and session optimization
        props.put(ConsumerConfig.CONNECTIONS_MAX_IDLE_MS_CONFIG, 600000); // 10
minutes
        props.put(ConsumerConfig.SESSION_TIMEOUT_MS_CONFIG, 30000); // 30 seconds
        props.put(ConsumerConfig.HEARTBEAT_INTERVAL_MS_CONFIG, 10000); // 10
seconds

        ConsumerFactory<String, String> consumerFactory = new
DefaultKafkaConsumerFactory<>(props);

        ConcurrentKafkaListenerContainerFactory<String, String> factory =
            new ConcurrentKafkaListenerContainerFactory<>();

        factory.setConsumerFactory(consumerFactory);
        factory.setBatchListener(true);

factory.getContainerProperties().setAckMode(ContainerProperties.AckMode.BATCH);
        factory.setConcurrency(5); // Higher concurrency for throughput

        // Configure batch processing timeout
        factory.getContainerProperties().setConsumerTaskExecutor(
            Executors.newCachedThreadPool(r -> {
                Thread thread = new Thread(r, "high-throughput-batch");
                thread.setDaemon(true);
                return thread;
            })
        );

        log.info("Configured high-throughput batch factory: maxRecords=2000,
concurrency=5");

        return factory;
    }

    /**
     * JSON batch listener factory for complex objects
     */
    @Bean("jsonBatchFactory")
    public ConcurrentKafkaListenerContainerFactory<String, Object>
jsonBatchFactory() {
        Map<String, Object> props = new HashMap<>();

        props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        props.put(ConsumerConfig.GROUP_ID_CONFIG, "json-batch-group");
        props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class);
        props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
JsonDeserializer.class);

        // JSON deserialization configuration
        props.put(JsonDeserializer.TRUSTED_PACKAGES, "com.example.*");
        props.put(JsonDeserializer.USE_TYPE_INFO_HEADERS, true);
        props.put(JsonDeserializer.VALUE_DEFAULT_TYPE, Object.class);

        // Batch configuration optimized for JSON
        props.put(ConsumerConfig.MAX_POLL_RECORDS_CONFIG, 1000);
        props.put(ConsumerConfig.FETCH_MIN_BYTES_CONFIG, 1024 * 100); // 100KB
        props.put(ConsumerConfig.FETCH_MAX_WAIT_MS_CONFIG, 200);

        ConsumerFactory<String, Object> consumerFactory = new
DefaultKafkaConsumerFactory<>(props);

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

        factory.setConsumerFactory(consumerFactory);
        factory.setBatchListener(true);

factory.getContainerProperties().setAckMode(ContainerProperties.AckMode.BATCH);
        factory.setConcurrency(4);

        log.info("Configured JSON batch factory: maxRecords=1000, JSON
deserialization enabled");

        return factory;
    }
}

/**
 * Comprehensive batch listener examples with different patterns
 */
@Component
@lombok.extern.slf4j.Slf4j
public class BatchMessageConsumers {

    /**
     * Basic batch listener - processes list of string messages
     */
    @KafkaListener(
        topics = "simple-logs",
        groupId = "simple-batch-group",
        containerFactory = "batchKafkaListenerContainerFactory"
    )
    public void processSimpleBatch(@Payload List<String> messages) {

        log.info("Processing simple batch: size={}", messages.size());

        // Process each message in the batch
        for (int i = 0; i < messages.size(); i++) {
            String message = messages.get(i);
            log.debug("Processing message {}: {}", i + 1, message);

            // Business logic for each message
            processLogMessage(message);
        }
        log.info("Completed simple batch processing: processed {} messages",
messages.size());
    }

    /**
     * Advanced batch listener with headers and metadata
     */
    @KafkaListener(
        topics = "application-logs",
        groupId = "advanced-batch-group",
        containerFactory = "batchKafkaListenerContainerFactory"
    )
    public void processAdvancedBatch(
            @Payload List<String> messages,
            @Header(KafkaHeaders.RECEIVED_PARTITION) List<Integer> partitions,
            @Header(KafkaHeaders.OFFSET) List<Long> offsets,
            @Header(KafkaHeaders.RECEIVED_TIMESTAMP) List<Long> timestamps,
            @Header(KafkaHeaders.RECEIVED_TOPIC) List<String> topics,
            Acknowledgment acknowledgment) {

        log.info("Processing advanced batch: size={}, unique partitions={}",
            messages.size(), partitions.stream().distinct().count());

        try {
            // Process messages with metadata
            for (int i = 0; i < messages.size(); i++) {
                String message = messages.get(i);
                Integer partition = partitions.get(i);
                Long offset = offsets.get(i);
                Long timestamp = timestamps.get(i);

                log.debug("Processing message from partition {} at offset {}: {}",
                    partition, offset, message);

                // Business logic with metadata
                processLogWithMetadata(message, partition, offset, timestamp);
            }

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

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

        } catch (Exception e) {
            log.error("Error processing advanced batch", e);
            throw e; // Will trigger error handler
        }
    }

    /**
     * High-volume batch listener for log aggregation
     */
    @KafkaListener(
        topics = "high-volume-logs",
        groupId = "log-aggregation-group",
        containerFactory = "highThroughputBatchFactory"
    )
    public void processHighVolumeBatch(@Payload List<String> logMessages,
                                       @Header(KafkaHeaders.RECEIVED_PARTITION)
List<Integer> partitions,
                                       Acknowledgment ack) {

        long startTime = System.currentTimeMillis();

        log.info("Processing high-volume batch: size={}, partitions={}",
            logMessages.size(),
partitions.stream().distinct().collect(Collectors.toList()));

        try {
            // Group messages by partition for efficient processing
            Map<Integer, List<String>> messagesByPartition =
groupMessagesByPartition(logMessages, partitions);

            // Process each partition's messages
            for (Map.Entry<Integer, List<String>> entry :
messagesByPartition.entrySet()) {
                Integer partition = entry.getKey();
                List<String> partitionMessages = entry.getValue();

                log.debug("Processing {} messages from partition {}",
partitionMessages.size(), partition);

                // Batch process messages from each partition
                processPartitionBatch(partition, partitionMessages);
            }

            // Acknowledge successful processing
            ack.acknowledge();

            long processingTime = System.currentTimeMillis() - startTime;
            double throughput = logMessages.size() / (processingTime / 1000.0);

            log.info("High-volume batch completed: {} messages in {}ms,
throughput: {:.2f} msg/sec",
                logMessages.size(), processingTime, throughput);

        } catch (Exception e) {
            log.error("Error processing high-volume batch: size={}",
logMessages.size(), e);
            throw e;
        }
    }

    /**
     * ConsumerRecord batch listener for full record access
     */
    @KafkaListener(
        topics = "detailed-logs",
        groupId = "consumer-record-batch-group",
        containerFactory = "batchKafkaListenerContainerFactory"
    )
    public void processConsumerRecordBatch(List<ConsumerRecord<String, String>>
records,
                                         Acknowledgment ack) {

        log.info("Processing ConsumerRecord batch: size={}", records.size());

        try {
            // Process each ConsumerRecord with full access to metadata
            for (ConsumerRecord<String, String> record : records) {
                log.debug("Processing record: topic={}, partition={}, offset={},
timestamp={}, key={}",
                    record.topic(), record.partition(), record.offset(),
record.timestamp(), record.key());

                // Access headers
                record.headers().forEach(header -> {
                    log.debug("Header: {}={}", header.key(), new
String(header.value()));
                });

                // Business logic with full record context
                processDetailedRecord(record);
            }

            ack.acknowledge();

            log.info("ConsumerRecord batch processing completed: {} records",
records.size());

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

    /**
     * ConsumerRecords batch listener for partition-aware processing
     */
    @KafkaListener(
        topics = "partitioned-logs",
        groupId = "consumer-records-batch-group",
        containerFactory = "batchKafkaListenerContainerFactory"
    )
    public void processConsumerRecordsBatch(ConsumerRecords<String, String>
consumerRecords,
                                          Acknowledgment ack) {

        log.info("Processing ConsumerRecords batch: total records={}, partitions=
{}",
            consumerRecords.count(), consumerRecords.partitions().size());

        try {
            // Process records by partition for optimal performance
            for (TopicPartition partition : consumerRecords.partitions()) {
                List<ConsumerRecord<String, String>> partitionRecords =
consumerRecords.records(partition);

                log.debug("Processing partition {}: {} records", partition,
partitionRecords.size());

                // Process partition records as a group
                processPartitionRecords(partition, partitionRecords);
            }

            ack.acknowledge();

            log.info("ConsumerRecords batch processing completed: {} total
records", consumerRecords.count());

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

    /**
     * JSON object batch listener for complex message types
     */
    @KafkaListener(
        topics = "json-events",
        groupId = "json-batch-group",
        containerFactory = "jsonBatchFactory"
    )
    public void processJsonBatch(@Payload List<EventData> events,
                                 @Header(KafkaHeaders.RECEIVED_MESSAGE_KEY)
List<String> keys,
                                 Acknowledgment ack) {

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

        try {
            // Group events by type for efficient processing
            Map<String, List<EventData>> eventsByType = events.stream()
                .collect(Collectors.groupingBy(EventData::getEventType));

            // Process each event type separately
            for (Map.Entry<String, List<EventData>> entry :
eventsByType.entrySet()) {
                String eventType = entry.getKey();
                List<EventData> typeEvents = entry.getValue();

                log.debug("Processing {} events of type: {}", typeEvents.size(),
eventType);
```

```
                // Type-specific batch processing
                switch (eventType) {
                    case "USER_ACTION" -> processUserActionBatch(typeEvents);
                    case "SYSTEM_EVENT" -> processSystemEventBatch(typeEvents);
                    case "ERROR_EVENT" -> processErrorEventBatch(typeEvents);
                    default -> processUnknownEventBatch(typeEvents);
                }
            }

            ack.acknowledge();

            log.info("JSON batch processing completed: {} events processed",
events.size());

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

    /**
     * Using Spring Kafka 2.8+ batch annotation override
     */
    @KafkaListener(
        topics = "mixed-processing-topic",
        groupId = "mixed-batch-group",
        batch = "true" // Override factory setting
    )
    public void processMixedBatch(List<String> messages) {
        log.info("Processing mixed batch using annotation override: size={}",
messages.size());

        // Process batch
        for (String message : messages) {
            processMixedMessage(message);
        }
    }

    // Business logic methods
    private void processLogMessage(String message) {
        log.trace("Processing log message: {}", message);
        // Implementation would parse and process log entry
    }

    private void processLogWithMetadata(String message, Integer partition, Long
offset, Long timestamp) {
        log.trace("Processing log with metadata: partition={}, offset={},
timestamp={}, message={}",
            partition, offset, timestamp, message);
        // Implementation would use metadata for enhanced processing
    }

    private Map<Integer, List<String>> groupMessagesByPartition(List<String>
messages, List<Integer> partitions) {
        Map<Integer, List<String>> result = new HashMap<>();

        for (int i = 0; i < messages.size(); i++) {
            Integer partition = partitions.get(i);
            result.computeIfAbsent(partition, k -> new ArrayList<>
()).add(messages.get(i));
        }

        return result;
    }

    private void processPartitionBatch(Integer partition, List<String> messages) {
        log.debug("Processing partition {} batch: {} messages", partition,
messages.size());

        // Optimized processing for messages from same partition
        // Can maintain state/cache per partition
        for (String message : messages) {
            processLogMessage(message);
        }
    }

    private void processDetailedRecord(ConsumerRecord<String, String> record) {
        log.trace("Processing detailed record: {}", record.value());
        // Implementation would use full record context
    }

    private void processPartitionRecords(TopicPartition partition,
List<ConsumerRecord<String, String>> records) {
        log.debug("Processing records from partition {}: {} records", partition,
records.size());

        // Partition-aware processing logic
        for (ConsumerRecord<String, String> record : records) {
            processDetailedRecord(record);
        }
    }

    private void processUserActionBatch(List<EventData> events) {
        log.debug("Processing user action batch: {} events", events.size());
        // User action specific batch processing
    }

    private void processSystemEventBatch(List<EventData> events) {
        log.debug("Processing system event batch: {} events", events.size());
        // System event specific batch processing
    }

    private void processErrorEventBatch(List<EventData> events) {
        log.debug("Processing error event batch: {} events", events.size());
        // Error event specific batch processing
    }
    private void processUnknownEventBatch(List<EventData> events) {
        log.debug("Processing unknown event batch: {} events", events.size());
        // Unknown event handling
    }

    private void processMixedMessage(String message) {
        log.trace("Processing mixed message: {}", message);
        // Mixed processing logic
    }
}

// Supporting data structures
@lombok.Data
@lombok.Builder
@lombok.NoArgsConstructor
@lombok.AllArgsConstructor
class EventData {
    private String eventId;
    private String eventType;
    private String userId;
    private Map<String, Object> data;
    private java.time.Instant timestamp;
}
```

BatchMessageListenerContainer

Advanced Container Configuration and Custom Listeners

```
import org.springframework.kafka.listener.KafkaMessageListenerContainer;
import org.springframework.kafka.listener.ConcurrentMessageListenerContainer;
import org.springframework.kafka.listener.BatchMessageListener;
import org.springframework.kafka.listener.BatchAcknowledgingMessageListener;
import org.springframework.kafka.listener.BatchConsumerAwareMessageListener;
import org.springframework.kafka.listener.ContainerProperties;

import org.apache.kafka.clients.consumer.Consumer;
import org.apache.kafka.clients.consumer.ConsumerRecords;
import org.apache.kafka.common.TopicPartition;

/**
 * Advanced BatchMessageListenerContainer configuration and custom implementations
 */
@Configuration
@lombok.extern.slf4j.Slf4j
public class BatchMessageListenerContainerConfiguration {

    @Autowired
    private ConsumerFactory<String, String> batchConsumerFactory;

    /**
     * Basic BatchMessageListenerContainer setup
     */
    @Bean
    public ConcurrentMessageListenerContainer<String, String>
basicBatchContainer() {

        // Container properties configuration
        ContainerProperties containerProps = new ContainerProperties("basic-batch-
topic");
        containerProps.setMessageListener(new BasicBatchMessageListener());
        containerProps.setAckMode(ContainerProperties.AckMode.BATCH);

        // Create concurrent container
        ConcurrentMessageListenerContainer<String, String> container =
            new ConcurrentMessageListenerContainer<>(batchConsumerFactory,
containerProps);

        container.setConcurrency(3);
        container.setBeanName("basicBatchContainer");

        log.info("Created basic batch message listener container");

        return container;
    }

    /**
     * Advanced BatchMessageListenerContainer with custom listener
     */
    @Bean
    public ConcurrentMessageListenerContainer<String, String>
advancedBatchContainer() {

        ContainerProperties containerProps = new ContainerProperties("advanced-
batch-topic");
        containerProps.setMessageListener(new AdvancedBatchMessageListener());
        containerProps.setAckMode(ContainerProperties.AckMode.MANUAL_IMMEDIATE);

        // Advanced container configuration
        containerProps.setPollTimeout(Duration.ofMillis(1000));
        containerProps.setShutdownTimeout(Duration.ofSeconds(10));
        containerProps.setLogContainerConfig(true);

        // Consumer lifecycle management
        containerProps.setConsumerTaskExecutor(createBatchTaskExecutor());

        ConcurrentMessageListenerContainer<String, String> container =
            new ConcurrentMessageListenerContainer<>(batchConsumerFactory,
containerProps);

        container.setConcurrency(5);
        container.setBeanName("advancedBatchContainer");

        // Container lifecycle management
        container.setAutoStartup(true);
        log.info("Created advanced batch message listener container with custom
executor");

        return container;
    }

    /**
     * High-performance BatchMessageListenerContainer for analytics
     */
    @Bean
    public ConcurrentMessageListenerContainer<String, String>
analyticsBatchContainer() {

        ContainerProperties containerProps = new ContainerProperties("analytics-
events");
        containerProps.setMessageListener(new AnalyticsBatchMessageListener());
        containerProps.setAckMode(ContainerProperties.AckMode.BATCH);

        // High-performance configuration
        containerProps.setPollTimeout(Duration.ofMillis(100)); // Low latency
        containerProps.setIdleBetweenPolls(Duration.ofMillis(50));

        // Error handling
        containerProps.setDeliveryAttemptHeader(true);

        ConcurrentMessageListenerContainer<String, String> container =
            new ConcurrentMessageListenerContainer<>(batchConsumerFactory,
containerProps);

        container.setConcurrency(8); // High concurrency
        container.setBeanName("analyticsBatchContainer");

        log.info("Created high-performance analytics batch container:
concurrency=8");

        return container;
    }

    /**
     * Partition-aware BatchMessageListenerContainer
     */
    @Bean
    public ConcurrentMessageListenerContainer<String, String>
partitionAwareBatchContainer() {

        ContainerProperties containerProps = new ContainerProperties("partition-
aware-topic");
        containerProps.setMessageListener(new PartitionAwareBatchListener());
        containerProps.setAckMode(ContainerProperties.AckMode.MANUAL_IMMEDIATE);

        // Partition assignment configuration
        containerProps.setClientId("partition-aware-client");

        ConcurrentMessageListenerContainer<String, String> container =
            new ConcurrentMessageListenerContainer<>(batchConsumerFactory,
containerProps);

        container.setConcurrency(4);
        container.setBeanName("partitionAwareBatchContainer");

        log.info("Created partition-aware batch container");

        return container;
    }

    private TaskExecutor createBatchTaskExecutor() {
        ThreadPoolTaskExecutor executor = new ThreadPoolTaskExecutor();
        executor.setCorePoolSize(5);
        executor.setMaxPoolSize(20);
        executor.setQueueCapacity(1000);
        executor.setThreadNamePrefix("batch-consumer-");
        executor.setWaitForTasksToCompleteOnShutdown(true);
        executor.setAwaitTerminationSeconds(30);
        executor.initialize();
        return executor;
    }
}

/**
 * Basic batch message listener implementation
 */
@Component
@lombok.extern.slf4j.Slf4j
public class BasicBatchMessageListener implements BatchMessageListener<String,
String> {

    @Override
    public void onMessage(List<ConsumerRecord<String, String>> records) {
        log.info("BasicBatchMessageListener processing batch: size={}",
records.size());

        try {
            // Simple batch processing
            for (ConsumerRecord<String, String> record : records) {
                processBasicRecord(record);
            }

            log.info("Basic batch processing completed: {} records",
records.size());

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

    private void processBasicRecord(ConsumerRecord<String, String> record) {
        log.debug("Processing basic record: topic={}, partition={}, offset={}",
            record.topic(), record.partition(), record.offset());

        // Basic processing logic
        String value = record.value();
        // Process the record value
    }
}

/**
 * Advanced batch message listener with acknowledgment
 */
@Component
@lombok.extern.slf4j.Slf4j
public class AdvancedBatchMessageListener implements
BatchAcknowledgingMessageListener<String, String> {

    @Autowired
    private BatchProcessingService batchProcessingService;

    @Autowired
    private MeterRegistry meterRegistry;

    @Override
    public void onMessage(List<ConsumerRecord<String, String>> records,
Acknowledgment acknowledgment) {
        Timer.Sample sample = Timer.start(meterRegistry);

        log.info("AdvancedBatchMessageListener processing batch: size={}",
records.size());

        try {
            // Group records by topic for processing
            Map<String, List<ConsumerRecord<String, String>>> recordsByTopic =
records.stream()
                .collect(Collectors.groupingBy(ConsumerRecord::topic));

            // Process each topic's records
            for (Map.Entry<String, List<ConsumerRecord<String, String>>> entry :
recordsByTopic.entrySet()) {
                String topic = entry.getKey();
                List<ConsumerRecord<String, String>> topicRecords =
entry.getValue();

                log.debug("Processing {} records from topic: {}",
topicRecords.size(), topic);

                // Topic-specific processing
                batchProcessingService.processTopicBatch(topic, topicRecords);
            }

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

            // Update metrics
            meterRegistry.counter("batch.processed.success", "listener",
"advanced")
                .increment(records.size());

            log.info("Advanced batch processing completed and acknowledged: {}
records", records.size());

        } catch (Exception e) {
            log.error("Error in advanced batch processing", e);

            meterRegistry.counter("batch.processed.error", "listener", "advanced")
                .increment(records.size());

            throw e; // Don't acknowledge on error

        } finally {
            sample.stop(Timer.builder("batch.processing.duration")
                .tag("listener", "advanced")
                .register(meterRegistry));
        }
    }
}

/**
 * Analytics batch message listener optimized for high-volume data
 */
@Component
@lombok.extern.slf4j.Slf4j
public class AnalyticsBatchMessageListener implements BatchMessageListener<String,
String> {

    @Autowired
    private AnalyticsAggregator analyticsAggregator;

    private final Map<String, AtomicLong> processingMetrics = new
ConcurrentHashMap<>();

    @Override
    public void onMessage(List<ConsumerRecord<String, String>> records) {
        long startTime = System.currentTimeMillis();

        log.info("AnalyticsBatchMessageListener processing batch: size={}",
records.size());

        try {
            // Extract analytics data from records
            List<AnalyticsEvent> events = records.parallelStream()
                .map(this::extractAnalyticsEvent)
                .filter(Objects::nonNull)
                .collect(Collectors.toList());

            // Aggregate events by time window
            Map<String, List<AnalyticsEvent>> eventsByWindow = events.stream()
                .collect(Collectors.groupingBy(this::getTimeWindow));
            // Process each time window
            for (Map.Entry<String, List<AnalyticsEvent>> entry :
eventsByWindow.entrySet()) {
                String timeWindow = entry.getKey();
                List<AnalyticsEvent> windowEvents = entry.getValue();

                log.debug("Processing analytics window {}: {} events", timeWindow,
windowEvents.size());

                // Aggregate analytics for time window
                analyticsAggregator.aggregateEvents(timeWindow, windowEvents);
            }

            // Update processing metrics
            long processingTime = System.currentTimeMillis() - startTime;
            double throughput = records.size() / (processingTime / 1000.0);

            processingMetrics.computeIfAbsent("total_processed", k -> new
AtomicLong(0))
                .addAndGet(records.size());

            log.info("Analytics batch completed: {} events in {}ms, throughput:
{:.2f} events/sec",
                records.size(), processingTime, throughput);

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

    private AnalyticsEvent extractAnalyticsEvent(ConsumerRecord<String, String>
record) {
        try {
            // Parse analytics event from record
            return AnalyticsEvent.fromJson(record.value());
        } catch (Exception e) {
            log.warn("Failed to parse analytics event: {}", record.value(), e);
            return null;
        }
    }

    private String getTimeWindow(AnalyticsEvent event) {
        // Create 5-minute time windows
        long timestamp = event.getTimestamp().toEpochMilli();
        long windowStart = (timestamp / 300000) * 300000; // 5 minutes in millis
        return Instant.ofEpochMilli(windowStart).toString();
    }
}

/**
 * Partition-aware batch listener for ordered processing
 */
```

```
@Component
@lombok.extern.slf4j.Slf4j
public class PartitionAwareBatchListener implements
BatchConsumerAwareMessageListener<String, String> {

    @Autowired
    private OrderedProcessingService orderedProcessingService;

    // Maintain state per partition
    private final Map<TopicPartition, PartitionState> partitionStates = new
ConcurrentHashMap<>();

    @Override
    public void onMessage(List<ConsumerRecord<String, String>> records,
                         Acknowledgment acknowledgment,
                         Consumer<?, ?> consumer) {

        log.info("PartitionAwareBatchListener processing batch: size={}",
records.size());

        try {
            // Group records by partition
            Map<TopicPartition, List<ConsumerRecord<String, String>>>
recordsByPartition =
                records.stream().collect(Collectors.groupingBy(record ->
                    new TopicPartition(record.topic(), record.partition())));

            // Process each partition's records in order
            for (Map.Entry<TopicPartition, List<ConsumerRecord<String, String>>>
entry : recordsByPartition.entrySet()) {
                TopicPartition partition = entry.getKey();
                List<ConsumerRecord<String, String>> partitionRecords =
entry.getValue();

                log.debug("Processing partition {}: {} records", partition,
partitionRecords.size());

                // Get or create partition state
                PartitionState state = partitionStates.computeIfAbsent(partition,
                    k -> new PartitionState());

                // Process records maintaining order within partition
                processPartitionRecords(partition, partitionRecords, state,
consumer);
            }

            // Acknowledge all partitions
            acknowledgment.acknowledge();

            log.info("Partition-aware batch processing completed: {} partitions,
{} total records",
                recordsByPartition.size(), records.size());

        } catch (Exception e) {
            log.error("Error in partition-aware batch processing", e);
            throw e;
        }
    }

    private void processPartitionRecords(TopicPartition partition,
                                         List<ConsumerRecord<String, String>>
records,
                                         PartitionState state,
                                         Consumer<?, ?> consumer) {

        // Sort records by offset to ensure order
        records.sort(Comparator.comparing(ConsumerRecord::offset));

        for (ConsumerRecord<String, String> record : records) {
            // Validate offset order
            if (record.offset() <= state.getLastProcessedOffset()) {
                log.warn("Received record with offset {} <= last processed offset
{} for partition {}",
                    record.offset(), state.getLastProcessedOffset(), partition);
                continue;
            }

            // Process record with ordering guarantee
            orderedProcessingService.processOrdered(partition, record, state);

            // Update partition state
            state.setLastProcessedOffset(record.offset());
            state.incrementProcessedCount();
        }

        log.debug("Processed {} records for partition {}, last offset: {}",
            records.size(), partition, state.getLastProcessedOffset());
    }
}

// Supporting services and classes
@Service
@lombok.extern.slf4j.Slf4j
public class BatchProcessingService {

    public void processTopicBatch(String topic, List<ConsumerRecord<String,
String>> records) {
        log.debug("Processing batch for topic {}: {} records", topic,
records.size());

        // Topic-specific processing logic
        switch (topic) {
            case "user-events" -> processUserEventBatch(records);
            case "system-logs" -> processSystemLogBatch(records);
            case "metrics" -> processMetricsBatch(records);
            default -> processGenericBatch(records);
        }
    }
    private void processUserEventBatch(List<ConsumerRecord<String, String>>
records) {
        // User event specific batch processing
    }

    private void processSystemLogBatch(List<ConsumerRecord<String, String>>
records) {
        // System log specific batch processing
    }

    private void processMetricsBatch(List<ConsumerRecord<String, String>> records)
{
        // Metrics specific batch processing
    }

    private void processGenericBatch(List<ConsumerRecord<String, String>> records)
{
        // Generic batch processing
    }
}

@Service
@lombok.extern.slf4j.Slf4j
public class AnalyticsAggregator {

    private final Map<String, AnalyticsWindow> windows = new ConcurrentHashMap<>
();

    public void aggregateEvents(String timeWindow, List<AnalyticsEvent> events) {
        AnalyticsWindow window = windows.computeIfAbsent(timeWindow,
            k -> new AnalyticsWindow(timeWindow));

        // Aggregate events into window
        for (AnalyticsEvent event : events) {
            window.addEvent(event);
        }

        log.debug("Aggregated {} events for window {}", events.size(),
timeWindow);
    }
}

@Service
@lombok.extern.slf4j.Slf4j
public class OrderedProcessingService {

    public void processOrdered(TopicPartition partition,
                              ConsumerRecord<String, String> record,
                              PartitionState state) {

        log.trace("Processing ordered record: partition={}, offset={}", partition,
record.offset());
```

```
        // Ordered processing logic that maintains state per partition
        // This ensures messages from same partition are processed in order

        String value = record.value();
        // Process the record maintaining order
    }
}

// Supporting data structures
@lombok.Data
class PartitionState {
    private long lastProcessedOffset = -1;
    private long processedCount = 0;
    private Map<String, Object> state = new HashMap<>();
}

@lombok.Data
@lombok.Builder
@lombok.NoArgsConstructor
@lombok.AllArgsConstructor
class AnalyticsEvent {
    private String eventId;
    private String eventType;
    private String userId;
    private Map<String, Object> properties;
    private Instant timestamp;

    public static AnalyticsEvent fromJson(String json) {
        // JSON parsing implementation
        return new AnalyticsEvent();
    }
}

@lombok.Data
class AnalyticsWindow {
    private final String timeWindow;
    private final Map<String, Long> eventCounts = new HashMap<>();
    private final Map<String, Set<String>> uniqueUsers = new HashMap<>();
    private long totalEvents = 0;

    public AnalyticsWindow(String timeWindow) {
        this.timeWindow = timeWindow;
    }

    public void addEvent(AnalyticsEvent event) {
        totalEvents++;
        eventCounts.merge(event.getEventType(), 1L, Long::sum);
        uniqueUsers.computeIfAbsent(event.getEventType(), k -> new HashSet<>())
            .add(event.getUserId());
    }
}
```

This comprehensive section covers Batch Listeners with both @KafkaListener and BatchMessageListenerContainer approaches, showing extensive configuration options and real-world patterns. The guide continues with error handling in batches next.