⚙️ Consumer Configuration

Deserializers

Simple Explanation: Deserializers convert bytes from Kafka messages back into Java objects. Proper deserializer configuration is critical for handling various data formats and error scenarios, especially with poison pill messages that can crash consumers.

Why Deserializers Matter:

- **Data Integrity**: Ensure correct conversion from bytes to objects
- **Error Resilience**: Handle malformed or corrupted messages gracefully
- **Type Safety**: Support for different message formats and schemas
- **Performance**: Efficient deserialization for high-throughput scenarios

Advanced Deserializer Configuration and Error Handling

```java
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.kafka.core.ConsumerFactory;
import org.springframework.kafka.core.DefaultKafkaConsumerFactory;
import org.springframework.kafka.support.serializer.ErrorHandlingDeserializer;
import org.springframework.kafka.support.serializer.JsonDeserializer;

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

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

/**
 * Advanced deserializer configurations with comprehensive error handling
 */
@Configuration
@lombok.extern.slf4j.Slf4j
public class DeserializerConfiguration {

    /**
     * Error-handling JSON deserializer - handles poison pills gracefully
     */
    @Bean
    public ConsumerFactory<String, Object> errorHandlingJsonConsumerFactory() {
        Map<String, Object> props = new HashMap<>();

        // Basic consumer configuration
        props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        props.put(ConsumerConfig.GROUP_ID_CONFIG, "error-handling-group");

        // Error-handling deserializers
        props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
ErrorHandlingDeserializer.class);
        props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
ErrorHandlingDeserializer.class);

        // Delegate deserializers for actual deserialization
        props.put(ErrorHandlingDeserializer.KEY_DESERIALIZER_CLASS,
StringDeserializer.class);
        props.put(ErrorHandlingDeserializer.VALUE_DESERIALIZER_CLASS,
JsonDeserializer.class);

        // JSON deserializer specific configuration
        props.put(JsonDeserializer.TRUSTED_PACKAGES, "*");
        props.put(JsonDeserializer.VALUE_DEFAULT_TYPE, Object.class);
        props.put(JsonDeserializer.TYPE_MAPPINGS,
            "order:com.example.OrderEvent," +
            "payment:com.example.PaymentEvent," +
            "notification:com.example.NotificationEvent");

        return new DefaultKafkaConsumerFactory<>(props);
    }

    /**
     * Custom JSON deserializer with advanced ObjectMapper configuration
     */
    @Bean
    public ConsumerFactory<String, Object> customJsonConsumerFactory() {
        Map<String, Object> props = new HashMap<>();
        props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        props.put(ConsumerConfig.GROUP_ID_CONFIG, "custom-json-group");
        props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class);
        props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
JsonDeserializer.class);

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

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

objectMapper.configure(com.fasterxml.jackson.databind.DeserializationFeature.FAIL_ON
_UNKNOWN_PROPERTIES, false);

objectMapper.configure(com.fasterxml.jackson.databind.DeserializationFeature.ACCEPT_
EMPTY_STRING_AS_NULL_OBJECT, true);

        // Set custom ObjectMapper
        JsonDeserializer<Object> jsonDeserializer = new JsonDeserializer<>
(objectMapper);
        jsonDeserializer.addTrustedPackages("*");
        factory.setValueDeserializer(jsonDeserializer);

        return factory;
    }
    /**
     * Avro deserializer with Schema Registry integration
     */
    @Bean
    public ConsumerFactory<String, Object> avroConsumerFactory() {
        Map<String, Object> props = new HashMap<>();
        props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        props.put(ConsumerConfig.GROUP_ID_CONFIG, "avro-group");
        props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class);
        props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
            "io.confluent.kafka.serializers.KafkaAvroDeserializer");

        // Schema Registry configuration
        props.put("schema.registry.url", "http://localhost:8081");
        props.put("specific.avro.reader", true);

        return new DefaultKafkaConsumerFactory<>(props);
    }

    /**
     * Multi-type deserializer using delegation
     */
    @Bean
    public ConsumerFactory<String, Object> multiTypeConsumerFactory() {
        Map<String, Object> props = new HashMap<>();
        props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        props.put(ConsumerConfig.GROUP_ID_CONFIG, "multi-type-group");
        props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class);
        props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
DelegatingByTopicDeserializer.class);

        // Topic-based deserializer mapping
        props.put(DelegatingByTopicDeserializer.VALUE_SERIALIZATION_TOPIC_CONFIG,
            "json-.*:org.springframework.kafka.support.serializer.JsonDeserializer,"
+
            "avro-.*:io.confluent.kafka.serializers.KafkaAvroDeserializer," +
            "string-.*:org.apache.kafka.common.serialization.StringDeserializer");

        return new DefaultKafkaConsumerFactory<>(props);
    }

    /**
     * Custom deserializer implementation
     */
    public static class CustomBusinessEventDeserializer implements
Deserializer<BusinessEvent> {

        private final ObjectMapper objectMapper;

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

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

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

            try {
                // Custom deserialization logic
                Map<String, Object> rawData = objectMapper.readValue(data,
Map.class);

                return BusinessEvent.builder()
                    .eventId((String) rawData.get("event_id"))
                    .eventType((String) rawData.get("event_type"))
                    .payload((String) rawData.get("payload"))
                    .timestamp(Instant.ofEpochMilli((Long)
rawData.get("timestamp")))
                    .version((String) rawData.getOrDefault("version", "1.0"))
                    .build();

            } catch (Exception e) {
                log.error("Failed to deserialize BusinessEvent from topic: {}",
topic, e);
                throw new RuntimeException("Deserialization failed", e);
            }
        }

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

    /**
     * Tolerant deserializer that handles version mismatches
     */
    public static class VersionTolerantDeserializer implements
Deserializer<VersionedEvent> {

        private final ObjectMapper objectMapper;
        private final Map<String, Class<? extends VersionedEvent>> versionMapping;

        public VersionTolerantDeserializer() {
            this.objectMapper = new ObjectMapper();
            this.objectMapper.configure(

com.fasterxml.jackson.databind.DeserializationFeature.FAIL_ON_UNKNOWN_PROPERTIES,
false);
```

```
            // Version mapping for backward compatibility
            this.versionMapping = new HashMap<>();
            versionMapping.put("1.0", VersionedEventV1.class);
            versionMapping.put("2.0", VersionedEventV2.class);
            versionMapping.put("3.0", VersionedEventV3.class);
        }

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

            try {
                // First, read as generic map to extract version
                Map<String, Object> rawData = objectMapper.readValue(data,
Map.class);
                String version = (String) rawData.getOrDefault("version", "1.0");

                // Get appropriate class for version
                Class<? extends VersionedEvent> targetClass =
versionMapping.get(version);
                if (targetClass == null) {
                    log.warn("Unknown version {}, defaulting to latest", version);
                    targetClass = VersionedEventV3.class; // Default to latest
                }

                // Deserialize to specific version class
                return objectMapper.readValue(data, targetClass);

            } catch (Exception e) {
                log.error("Failed to deserialize VersionedEvent from topic: {}",
topic, e);

                // Fallback: try to deserialize as base class
                try {
                    return objectMapper.readValue(data, VersionedEventV1.class);
                } catch (Exception fallbackException) {
                    log.error("Fallback deserialization also failed",
fallbackException);
                    throw new RuntimeException("Complete deserialization failure",
e);
                }
            }
        }

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

    /**
     * Container factory with error-handling deserializers
     */
    @Bean
    public ConcurrentKafkaListenerContainerFactory<String, Object>
errorHandlingContainerFactory() {
        ConcurrentKafkaListenerContainerFactory<String, Object> factory =
            new ConcurrentKafkaListenerContainerFactory<>();

        factory.setConsumerFactory(errorHandlingJsonConsumerFactory());

        // Configure error handler to handle deserialization exceptions
        factory.setCommonErrorHandler(deserializationErrorHandler());

        return factory;
    }

    /**
     * Error handler specifically for deserialization errors
     */
    @Bean
    public CommonErrorHandler deserializationErrorHandler() {
        return new DefaultErrorHandler((record, exception) -> {
            // Handle deserialization failures
            log.error("Deserialization error for record: topic={}, partition={},
offset={}",
                record.topic(), record.partition(), record.offset());

            // Check if it's a deserialization exception
            if (exception.getCause() instanceof
org.springframework.kafka.support.serializer.DeserializationException) {
                handleDeserializationException(record,

(org.springframework.kafka.support.serializer.DeserializationException)
exception.getCause());
            } else {
                log.error("Non-deserialization error", exception);
            }
        });
    }

    private void handleDeserializationException(ConsumerRecord<?, ?> record,

org.springframework.kafka.support.serializer.DeserializationException ex) {

        log.error("Deserialization exception details: message={}, data={}",
            ex.getMessage(),
            ex.getData() != null ? new String(ex.getData()) : "null");

        // Could send to dead letter topic, log to database, etc.
        sendPoisonPillToDeadLetter(record, ex);
    }

    private void sendPoisonPillToDeadLetter(ConsumerRecord<?, ?> record, Exception
ex) {
        log.info("Sending poison pill to dead letter topic: {}", record.topic() +
".DLT");
```

```
            // Implementation would send to DLT
        }
    }

    /**
     * Deserializer examples and patterns
     */
    @Component
    @lombok.extern.slf4j.Slf4j
    public class DeserializerExamples {

        /**
         * Listener with error-handling deserializers
         * Gracefully handles poison pills
         */
        @KafkaListener(
            topics = "error-prone-topic",
            groupId = "resilient-consumer",
            containerFactory = "errorHandlingContainerFactory"
        )
        public void handleErrorProneMessages(ConsumerRecord<String, Object> record) {

            // Check for deserialization errors in headers
            DeserializationException keyError = checkForDeserializationError(record,
                ErrorHandlingDeserializer.KEY_DESERIALIZER_EXCEPTION_HEADER);
            DeserializationException valueError = checkForDeserializationError(record,
                ErrorHandlingDeserializer.VALUE_DESERIALIZER_EXCEPTION_HEADER);

            if (keyError != null) {
                log.error("Key deserialization failed: {}", keyError.getMessage());
                handleKeyDeserializationError(record, keyError);
                return;
            }

            if (valueError != null) {
                log.error("Value deserialization failed: {}", valueError.getMessage());
                handleValueDeserializationError(record, valueError);
                return;
            }

            // Normal processing
            processValidMessage(record);
        }

        /**
         * Type-safe message processing
         */
        @KafkaListener(
            topics = "business-events",
            groupId = "business-processor",
            containerFactory = "customJsonContainerFactory"
        )
        public void handleBusinessEvents(ConsumerRecord<String, Object> record) {
            Object value = record.value();
            if (value instanceof BusinessEvent businessEvent) {
                processBusinessEvent(businessEvent);
            } else if (value instanceof OrderEvent orderEvent) {
                processOrderEvent(orderEvent);
            } else {
                log.warn("Unknown message type: {}", value.getClass());
                handleUnknownMessageType(record);
            }
        }

        /**
         * Versioned event processing
         */
        @KafkaListener(
            topics = "versioned-events",
            groupId = "versioned-processor"
        )
        public void handleVersionedEvents(VersionedEvent event) {
            log.info("Processing versioned event: version={}, id={}",
                event.getVersion(), event.getEventId());

            // Version-specific processing
            switch (event.getVersion()) {
                case "1.0" -> processV1Event((VersionedEventV1) event);
                case "2.0" -> processV2Event((VersionedEventV2) event);
                case "3.0" -> processV3Event((VersionedEventV3) event);
                default -> {
                    log.warn("Unsupported version: {}", event.getVersion());
                    processAsLatestVersion(event);
                }
            }
        }

        /**
         * Multi-format message processing
         */
        @KafkaListener(
            topics = {"json-messages", "avro-messages", "string-messages"},
            groupId = "multi-format-processor",
            containerFactory = "multiTypeContainerFactory"
        )
        public void handleMultiFormatMessages(
                @Payload Object message,
                @Header(KafkaHeaders.RECEIVED_TOPIC) String topic) {

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

            if (topic.startsWith("json-")) {
                processJsonMessage(message);
            } else if (topic.startsWith("avro-")) {
                processAvroMessage(message);
            } else if (topic.startsWith("string-")) {
                processStringMessage((String) message);
            } else {
                log.warn("Unknown topic format: {}", topic);
            }
        }

        // Helper methods
        private DeserializationException
    checkForDeserializationError(ConsumerRecord<String, Object> record,
                                                                    String headerName) {
            Header header = record.headers().lastHeader(headerName);
            if (header != null) {
                try {
                    return SerializationUtils.deserialize(header.value());
                } catch (Exception e) {
                    log.error("Failed to deserialize error header", e);
                }
            }
            return null;
        }

        private void handleKeyDeserializationError(ConsumerRecord<String, Object>
    record,
                                                          DeserializationException error) {
            log.error("Handling key deserialization error: {}", error.getMessage());
            // Could implement specific key error handling
        }

        private void handleValueDeserializationError(ConsumerRecord<String, Object>
    record,
                                                            DeserializationException error) {
            log.error("Handling value deserialization error: {}", error.getMessage());
            // Could implement specific value error handling
        }

        private void processValidMessage(ConsumerRecord<String, Object> record) {
            log.debug("Processing valid message: key={}, value={}",
                record.key(), record.value());
        }

        private void processBusinessEvent(BusinessEvent event) {
            log.debug("Processing business event: {}", event.getEventId());
        }

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

        private void handleUnknownMessageType(ConsumerRecord<String, Object> record) {
            log.warn("Handling unknown message type from topic: {}", record.topic());
        }

        private void processV1Event(VersionedEventV1 event) {
            log.debug("Processing V1 event: {}", event.getEventId());
        }

        private void processV2Event(VersionedEventV2 event) {
            log.debug("Processing V2 event: {}", event.getEventId());
        }

        private void processV3Event(VersionedEventV3 event) {
            log.debug("Processing V3 event: {}", event.getEventId());
        }

        private void processAsLatestVersion(VersionedEvent event) {
            log.debug("Processing as latest version: {}", event.getEventId());
        }

        private void processJsonMessage(Object message) {
            log.debug("Processing JSON message: {}",
    message.getClass().getSimpleName());
        }

        private void processAvroMessage(Object message) {
            log.debug("Processing Avro message: {}",
    message.getClass().getSimpleName());
        }

        private void processStringMessage(String message) {
            log.debug("Processing string message: {}", message);
        }
    }

    // Domain objects for deserializer examples
    @lombok.Data
    @lombok.Builder
    @lombok.NoArgsConstructor
    @lombok.AllArgsConstructor
    class BusinessEvent {
        private String eventId;
        private String eventType;
        private String payload;
        private Instant timestamp;
        private String version;
    }

    @lombok.Data
    @lombok.NoArgsConstructor
    @lombok.AllArgsConstructor
    abstract class VersionedEvent {
        protected String eventId;
        protected String version;
        protected Instant timestamp;

        public abstract String getVersion();
        public abstract String getEventId();
    }

    @lombok.Data
    @lombok.EqualsAndHashCode(callSuper = true)
    @lombok.NoArgsConstructor
    @lombok.AllArgsConstructor
    class VersionedEventV1 extends VersionedEvent {
        private String data;

        @Override
        public String getVersion() { return "1.0"; }
    }

    @lombok.Data
    @lombok.EqualsAndHashCode(callSuper = true)
    @lombok.NoArgsConstructor
    @lombok.AllArgsConstructor
    class VersionedEventV2 extends VersionedEvent {
        private String data;
        private Map<String, String> metadata;

        @Override
        public String getVersion() { return "2.0"; }
    }

    @lombok.Data
    @lombok.EqualsAndHashCode(callSuper = true)
    @lombok.NoArgsConstructor
    @lombok.AllArgsConstructor
    class VersionedEventV3 extends VersionedEvent {
        private String data;
        private Map<String, String> metadata;
        private List<String> tags;

        @Override
        public String getVersion() { return "3.0"; }
    }
```

Rebalance Listeners

Simple Explanation: Rebalance listeners allow applications to hook into the consumer group rebalancing process, enabling custom offset management, resource cleanup, and graceful handling of partition reassignments.

Why Rebalance Listeners are Important:

- **Custom Offset Management**: Store and retrieve offsets from external systems
- **Resource Cleanup**: Clean up resources when partitions are revoked
- **Graceful Shutdown**: Properly handle partition reassignment during scaling
- **Monitoring**: Track rebalancing events for operational visibility

Advanced Rebalance Listener Implementation

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.kafka.config.ConcurrentKafkaListenerContainerFactory;
import org.springframework.kafka.listener.ConsumerAwareRebalanceListener;
import org.springframework.kafka.listener.ContainerProperties;
import org.apache.kafka.clients.consumer.Consumer;
import org.apache.kafka.clients.consumer.ConsumerRebalanceListener;
import org.apache.kafka.clients.consumer.OffsetAndMetadata;
import org.apache.kafka.common.TopicPartition;

import java.util.Collection;
import java.util.Map;
import java.util.concurrent.ConcurrentHashMap;

/**
 * Advanced rebalance listener configurations and implementations
 */
@Configuration
@lombok.extern.slf4j.Slf4j
public class RebalanceListenerConfiguration {

    /**
     * Container factory with custom rebalance listener
     */
    @Bean
    public ConcurrentKafkaListenerContainerFactory<String, Object>
rebalanceAwareContainerFactory() {
        ConcurrentKafkaListenerContainerFactory<String, Object> factory =
            new ConcurrentKafkaListenerContainerFactory<>();

        factory.setConsumerFactory(consumerFactory());

        // Set custom rebalance listener
        factory.getContainerProperties().setConsumerRebalanceListener(
            new ComprehensiveRebalanceListener());

        // Manual acknowledgment for precise offset control

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

        return factory;
    }

    /**
     * Container factory for external offset management
     */
    @Bean
    public ConcurrentKafkaListenerContainerFactory<String, Object>
externalOffsetContainerFactory() {
        ConcurrentKafkaListenerContainerFactory<String, Object> factory =
            new ConcurrentKafkaListenerContainerFactory<>();

        factory.setConsumerFactory(consumerFactory());

        // External offset management rebalance listener
        factory.getContainerProperties().setConsumerRebalanceListener(
            new ExternalOffsetManagementListener());

        // Disable auto-commit for external offset management
```

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

        return factory;
    }

    /**
     * Comprehensive rebalance listener with full lifecycle management
     */
    public static class ComprehensiveRebalanceListener implements
ConsumerAwareRebalanceListener {

        private final Map<TopicPartition, OffsetAndMetadata> currentOffsets = new
ConcurrentHashMap<>();
        private final Map<TopicPartition, Long> partitionStartTimes = new
ConcurrentHashMap<>();
        private final Map<TopicPartition, Long> messagesCounts = new
ConcurrentHashMap<>();

        @Override
        public void onPartitionsRevokedBeforeCommit(Consumer<?, ?> consumer,
                                                   Collection<TopicPartition>
partitions) {

            log.info("🔄 Partitions being revoked BEFORE commit: {}", partitions);

            // Acknowledge any pending manual acknowledgments
            // This is crucial for manual acknowledgment modes
            for (TopicPartition partition : partitions) {
                long processingDuration = System.currentTimeMillis() -
                    partitionStartTimes.getOrDefault(partition,
System.currentTimeMillis());

                log.info("Partition {} was owned for {}ms, processed {} messages",
                    partition, processingDuration,
messagesCounts.getOrDefault(partition, 0L));
            }

            // Save current state before commit
            savePartitionState(partitions);
        }

        @Override
        public void onPartitionsRevokedAfterCommit(Consumer<?, ?> consumer,
                                                 Collection<TopicPartition>
partitions) {

            log.info("🔄 Partitions revoked AFTER commit: {}", partitions);

            // Clean up resources after commit
            for (TopicPartition partition : partitions) {
                currentOffsets.remove(partition);
                partitionStartTimes.remove(partition);
                messagesCounts.remove(partition);

                // Clean up any partition-specific resources
                cleanupPartitionResources(partition);
            }

            // Perform any final cleanup
            performPostRevokeCleanup(partitions);
        }

        @Override
        public void onPartitionsAssigned(Consumer<?, ?> consumer,
                                        Collection<TopicPartition> partitions) {

            log.info("✅ Partitions assigned: {}", partitions);

            // Initialize partition tracking
            long currentTime = System.currentTimeMillis();
            for (TopicPartition partition : partitions) {
                partitionStartTimes.put(partition, currentTime);
                messagesCounts.put(partition, 0L);

                // Initialize partition-specific resources
                initializePartitionResources(partition);

                // Seek to custom offset if needed
                seekToCustomOffset(consumer, partition);
            }

            // Log partition assignment details
            logPartitionAssignmentDetails(consumer, partitions);
        }

        @Override
        public void onPartitionsLost(Consumer<?, ?> consumer,
                                    Collection<TopicPartition> partitions) {

            log.warn("⚠️ Partitions lost (unclean): {}", partitions);

            // Handle partition loss - don't commit offsets as partitions are
already reassigned
            for (TopicPartition partition : partitions) {
                currentOffsets.remove(partition);
                partitionStartTimes.remove(partition);
                messagesCounts.remove(partition);

                // Emergency cleanup for lost partitions
                handlePartitionLoss(partition);
            }

            // Alert monitoring systems
            alertPartitionLoss(partitions);
        }

        // Helper methods
        private void savePartitionState(Collection<TopicPartition> partitions) {
            for (TopicPartition partition : partitions) {
                log.debug("Saving state for partition: {}", partition);
                // Could save to database, Redis, etc.
            }
        }

        private void cleanupPartitionResources(TopicPartition partition) {
            log.debug("Cleaning up resources for partition: {}", partition);
            // Close connections, clear caches, etc.
        }

        private void performPostRevokeCleanup(Collection<TopicPartition> partitions)
{
            log.debug("Performing post-revoke cleanup for {} partitions",
partitions.size());
            // Global cleanup after partition revocation
        }

        private void initializePartitionResources(TopicPartition partition) {
            log.debug("Initializing resources for partition: {}", partition);
            // Initialize connections, caches, etc.
        }

        private void seekToCustomOffset(Consumer<?, ?> consumer, TopicPartition
partition) {
            // Example: seek to custom offset from external store
            Long customOffset = loadCustomOffset(partition);
            if (customOffset != null) {
                consumer.seek(partition, customOffset);
                log.info("Seeked partition {} to custom offset: {}", partition,
customOffset);
            }
        }

        private Long loadCustomOffset(TopicPartition partition) {
            // Load offset from external store (database, Redis, etc.)
            return null; // Placeholder
        }

        private void logPartitionAssignmentDetails(Consumer<?, ?> consumer,
                                                 Collection<TopicPartition>
partitions) {
            for (TopicPartition partition : partitions) {
                long currentPosition = consumer.position(partition);
                log.info("Assigned partition {} at position: {}", partition,
currentPosition);
            }
        }

        private void handlePartitionLoss(TopicPartition partition) {
            log.error("Handling partition loss: {}", partition);
            // Emergency cleanup, alerting, etc.
        }

        private void alertPartitionLoss(Collection<TopicPartition> partitions) {
            log.error("🚨 ALERT: Lost {} partitions: {}", partitions.size(),
partitions);
```

```
            // Send alerts to monitoring systems
        }

        // Public method to update message count (called by listeners)
        public void incrementMessageCount(TopicPartition partition) {
            messagesCounts.compute(partition, (k, v) -> (v == null) ? 1 : v + 1);
        }
    }

    /**
     * External offset management rebalance listener
     * Stores offsets in external system (database, Redis, etc.)
     */
    public static class ExternalOffsetManagementListener implements
ConsumerAwareRebalanceListener {

        private final OffsetRepository offsetRepository;

        public ExternalOffsetManagementListener() {
            this.offsetRepository = new DatabaseOffsetRepository(); // Example
implementation
        }

        @Override
        public void onPartitionsRevokedBeforeCommit(Consumer<?, ?> consumer,
                                                    Collection<TopicPartition>
partitions) {

            log.info("Saving offsets to external store before commit: {}",
partitions);

            // Save current offsets to external store
            for (TopicPartition partition : partitions) {
                long currentOffset = consumer.position(partition);
                offsetRepository.saveOffset(partition, currentOffset);

                log.debug("Saved offset for {}: {}", partition, currentOffset);
            }
        }

        @Override
        public void onPartitionsRevokedAfterCommit(Consumer<?, ?> consumer,
                                                   Collection<TopicPartition>
partitions) {
            log.info("Partitions revoked after commit (external offset): {}",
partitions);
            // Additional cleanup if needed
        }

        @Override
        public void onPartitionsAssigned(Consumer<?, ?> consumer,
                                         Collection<TopicPartition> partitions) {

            log.info("Loading offsets from external store for assigned partitions:
{}", partitions);
```

```
            // Load offsets from external store and seek
            for (TopicPartition partition : partitions) {
                Long savedOffset = offsetRepository.loadOffset(partition);

                if (savedOffset != null) {
                    consumer.seek(partition, savedOffset);
                    log.info("Restored partition {} to saved offset: {}", partition,
savedOffset);
                } else {
                    // No saved offset, use default behavior
                    log.info("No saved offset for partition {}, using default",
partition);
                }
            }
        }

        @Override
        public void onPartitionsLost(Consumer<?, ?> consumer,
                                     Collection<TopicPartition> partitions) {
            log.warn("Partitions lost (external offset management): {}",
partitions);
            // Don't save offsets for lost partitions
        }
    }

    /**
     * Graceful shutdown rebalance listener
     * Ensures clean shutdown during rebalancing
     */
    public static class GracefulShutdownRebalanceListener implements
ConsumerAwareRebalanceListener {

        private final AtomicBoolean isShuttingDown = new AtomicBoolean(false);
        private final CountDownLatch shutdownLatch = new CountDownLatch(1);

        @Override
        public void onPartitionsRevokedBeforeCommit(Consumer<?, ?> consumer,
                                                    Collection<TopicPartition>
partitions) {

            if (isShuttingDown.get()) {
                log.info("Graceful shutdown in progress, waiting for processing to
complete");

                try {
                    // Wait for current processing to complete
                    boolean completed = shutdownLatch.await(30, TimeUnit.SECONDS);
                    if (!completed) {
                        log.warn("Graceful shutdown timeout, forcing shutdown");
                    }
                } catch (InterruptedException e) {
                    Thread.currentThread().interrupt();
                    log.warn("Shutdown interrupted");
                }
            }

            log.info("Committing final offsets before partition revoke: {}",
partitions);

            // Ensure all processing is complete before committing
            for (TopicPartition partition : partitions) {
                finalizePartitionProcessing(partition);
            }
        }

        @Override
        public void onPartitionsRevokedAfterCommit(Consumer<?, ?> consumer,
                                                   Collection<TopicPartition>
partitions) {
            log.info("Graceful revoke completed for partitions: {}", partitions);
        }

        @Override
        public void onPartitionsAssigned(Consumer<?, ?> consumer,
                                         Collection<TopicPartition> partitions) {
            log.info("Partitions assigned during graceful operation: {}",
partitions);
        }

        @Override
        public void onPartitionsLost(Consumer<?, ?> consumer,
                                     Collection<TopicPartition> partitions) {
            log.warn("Partitions lost during graceful operation: {}", partitions);
        }

        public void initiateShutdown() {
            isShuttingDown.set(true);
        }

        public void markProcessingComplete() {
            shutdownLatch.countDown();
        }

        private void finalizePartitionProcessing(TopicPartition partition) {
            log.debug("Finalizing processing for partition: {}", partition);
            // Ensure all in-flight processing for partition is complete
        }
    }

    // Supporting interfaces and classes
    interface OffsetRepository {
        void saveOffset(TopicPartition partition, long offset);
        Long loadOffset(TopicPartition partition);
    }

    static class DatabaseOffsetRepository implements OffsetRepository {

        @Override
        public void saveOffset(TopicPartition partition, long offset) {
            log.debug("Saving offset to database: {}={}", partition, offset);
            // Database implementation
        }

        @Override
        public Long loadOffset(TopicPartition partition) {
            log.debug("Loading offset from database: {}", partition);
            // Database implementation
            return null; // Placeholder
        }
    }

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

/**
 * Rebalance-aware listener examples
 */
@Component
@lombok.extern.slf4j.Slf4j
public class RebalanceAwareListeners {

    @Autowired
    private ComprehensiveRebalanceListener rebalanceListener;

    /**
     * Listener that tracks rebalancing events
     */
    @KafkaListener(
        topics = "rebalance-aware-topic",
        groupId = "rebalance-aware-group",
        containerFactory = "rebalanceAwareContainerFactory"
    )
    public void handleRebalanceAwareMessages(
            @Payload EventMessage message,
            @Header(KafkaHeaders.RECEIVED_PARTITION_ID) int partition,
            @Header(KafkaHeaders.RECEIVED_TOPIC) String topic,
            Acknowledgment ack) {

        log.info("Processing rebalance-aware message: partition={}, eventId={}",
            partition, message.getEventId());

        try {
            processMessage(message);

            // Update message count for rebalance tracking
            TopicPartition topicPartition = new TopicPartition(topic, partition);
            rebalanceListener.incrementMessageCount(topicPartition);

            // Manual acknowledgment
            ack.acknowledge();
        } catch (Exception e) {
            log.error("Failed to process rebalance-aware message: {}",
message.getEventId(), e);
            throw e;
        }
    }

    /**
     * Listener with external offset management
     */
    @KafkaListener(
        topics = "external-offset-topic",
        groupId = "external-offset-group",
        containerFactory = "externalOffsetContainerFactory"
    )
    public void handleExternalOffsetMessages(@Payload EventMessage message,
                                             Acknowledgment ack) {

        log.info("Processing message with external offset management: {}",
message.getEventId());

        try {
            processMessage(message);

            // Manual acknowledgment - offset will be saved externally
            ack.acknowledge();

        } catch (Exception e) {
            log.error("Failed to process external offset message: {}",
message.getEventId(), e);
            throw e;
        }
    }

    private void processMessage(EventMessage message) {
        log.debug("Processing message: {}", message.getEventId());
        // Business logic here
    }
}
```

Error Handling Strategies

Simple Explanation: Error handling strategies in Spring Kafka determine how the consumer deals with processing failures, including retry mechanisms, dead letter topics, and recovery procedures to ensure reliable message processing.

Types of Errors:

- **Transient Errors**: Network issues, temporary service unavailability (retryable)
- **Poison Pills**: Malformed messages that always fail (non-retryable)
- **Business Logic Errors**: Application-specific failures (may or may not be retryable)
- **Deserialization Errors**: Failed message parsing (handled by ErrorHandlingDeserializer)

Comprehensive Error Handling Implementation

```java
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.kafka.config.ConcurrentKafkaListenerContainerFactory;
import org.springframework.kafka.core.KafkaTemplate;
import org.springframework.kafka.listener.CommonErrorHandler;
import org.springframework.kafka.listener.DefaultErrorHandler;
import org.springframework.kafka.listener.DeadLetterPublishingRecoverer;
import org.springframework.kafka.support.ExponentialBackOffWithMaxRetries;

import org.springframework.util.backoff.BackOff;
import org.springframework.util.backoff.ExponentialBackOff;
import org.springframework.util.backoff.FixedBackOff;

import java.util.List;
import java.util.function.BiFunction;

/**
 * Advanced error handling configurations and strategies
 */
@Configuration
@lombok.extern.slf4j.Slf4j
public class ErrorHandlingConfiguration {

    @Autowired
    private KafkaTemplate<String, Object> kafkaTemplate;

    /**
     * Default error handler with retry and recovery
     */
    @Bean
    public CommonErrorHandler defaultErrorHandler() {

        // Dead letter publishing recoverer
        DeadLetterPublishingRecoverer recoverer = new DeadLetterPublishingRecoverer(
            kafkaTemplate,
            this::deadLetterTopicResolver
        );

        // Exponential backoff: start at 1s, max 10s, multiplier 2.0, max 3 attempts
        ExponentialBackOff backOff = new ExponentialBackOff(1000L, 2.0);
        backOff.setMaxElapsedTime(30000L); // Max 30 seconds total

        DefaultErrorHandler errorHandler = new DefaultErrorHandler(recoverer,
backOff);

        // Configure non-retryable exceptions
        errorHandler.addNotRetryableExceptions(
            IllegalArgumentException.class,
            NullPointerException.class,

org.springframework.kafka.support.serializer.DeserializationException.class
        );
        // Configure retryable exceptions
        errorHandler.addRetryableExceptions(
            java.util.concurrent.TimeoutException.class,
            org.springframework.dao.TransientDataAccessException.class
        );

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

        return errorHandler;
    }

    /**
     * Fixed retry error handler for predictable retry behavior
     */
    @Bean
    public CommonErrorHandler fixedRetryErrorHandler() {

        // Custom recoverer with detailed logging
        DeadLetterPublishingRecoverer recoverer = new DeadLetterPublishingRecoverer(
            kafkaTemplate,
            (record, ex) -> {
                log.error("Sending to DLT after fixed retries: topic={}, key={}, 
error={}",
                    record.topic(), record.key(), ex.getMessage());
                return new TopicPartition(record.topic() + ".DLT", 
record.partition());
            }
        );

        // Fixed backoff: 2 seconds between retries, max 3 attempts
        FixedBackOff backOff = new FixedBackOff(2000L, 3L);

        DefaultErrorHandler errorHandler = new DefaultErrorHandler(recoverer, 
backOff);

        // Add exception classification
        errorHandler.addNotRetryableExceptions(ValidationException.class);

        return errorHandler;
    }

    /**
     * Custom error handler with business logic integration
     */
    @Bean
    public CommonErrorHandler businessLogicErrorHandler() {

        // Custom recoverer with business logic
        BusinessAwareRecoverer recoverer = new 
BusinessAwareRecoverer(kafkaTemplate);

        // Exponential backoff with max retries
        ExponentialBackOffWithMaxRetries backOff = new 
```

```
ExponentialBackOffWithMaxRetries(3);
        backOff.setInitialInterval(500L);
        backOff.setMultiplier(2.0);
        backOff.setMaxInterval(5000L);

        DefaultErrorHandler errorHandler = new DefaultErrorHandler(recoverer,
backOff);

        // Business-specific exception handling
        errorHandler.addNotRetryableExceptions(
            PermanentBusinessException.class,
            DataValidationException.class
        );

        errorHandler.addRetryableExceptions(
            TemporaryBusinessException.class,
            ExternalServiceException.class
        );

        return errorHandler;
    }

    /**
     * No-retry error handler for fire-and-forget scenarios
     */
    @Bean
    public CommonErrorHandler noRetryErrorHandler() {

        // Log-only recoverer
        ConsumerRecordRecoverer logOnlyRecoverer = (record, exception) -> {
            log.error("Message processing failed (no retry): topic={}, partition={},
offset={}, error={}",
                record.topic(), record.partition(), record.offset(),
exception.getMessage());
        };

        // No retries - immediate recovery
        DefaultErrorHandler errorHandler = new DefaultErrorHandler(logOnlyRecoverer,
new FixedBackOff(0L, 0L));

        return errorHandler;
    }

    /**
     * Container factories with different error handling strategies
     */
    @Bean
    public ConcurrentKafkaListenerContainerFactory<String, Object>
resilientContainerFactory() {
        ConcurrentKafkaListenerContainerFactory<String, Object> factory =
            new ConcurrentKafkaListenerContainerFactory<>();

        factory.setConsumerFactory(consumerFactory());
        factory.setCommonErrorHandler(defaultErrorHandler());
        return factory;
    }

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

        factory.setConsumerFactory(consumerFactory());
        factory.setCommonErrorHandler(fixedRetryErrorHandler());

        return factory;
    }

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

        factory.setConsumerFactory(consumerFactory());
        factory.setCommonErrorHandler(businessLogicErrorHandler());

        return factory;
    }

    // Helper methods and beans
    private TopicPartition deadLetterTopicResolver(ConsumerRecord<?, ?> record,
Exception ex) {
        // Determine DLT based on exception type
        String dltTopic;

        if (ex instanceof ValidationException) {
            dltTopic = record.topic() + ".validation.DLT";
        } else if (ex instanceof BusinessException) {
            dltTopic = record.topic() + ".business.DLT";
        } else {
            dltTopic = record.topic() + ".DLT";
        }

        log.info("Routing failed message to DLT: {} -> {}", record.topic(),
dltTopic);
        return new TopicPartition(dltTopic, record.partition());
    }

    @Bean
    public RetryListener retryListener() {
        return new RetryListener() {
            @Override
            public void failedDelivery(ConsumerRecord<?, ?> record, Exception ex,
int deliveryAttempt) {
                log.warn("Delivery attempt {} failed for record: topic={},
partition={}, offset={}, error={}",
                    deliveryAttempt, record.topic(), record.partition(),
record.offset(), ex.getMessage());

                // Update retry metrics
                updateRetryMetrics(record.topic(), deliveryAttempt);
            }

            @Override
            public void recovered(ConsumerRecord<?, ?> record, Exception ex) {
                log.info("Record recovered after retries: topic={}, partition={},
offset={}",
                    record.topic(), record.partition(), record.offset());

                // Update recovery metrics
                updateRecoveryMetrics(record.topic());
            }

            @Override
            public void recoveryFailed(ConsumerRecord<?, ?> record, Exception
original, Exception failure) {
                log.error("Recovery failed for record: topic={}, partition={},
offset={}, original={}, recovery={}",
                    record.topic(), record.partition(), record.offset(),
                    original.getMessage(), failure.getMessage());

                // Update failure metrics
                updateFailureMetrics(record.topic());
            }
        };
    }

    /**
     * Custom recoverer with business logic awareness
     */
    public static class BusinessAwareRecoverer implements ConsumerRecordRecoverer {

        private final KafkaTemplate<String, Object> kafkaTemplate;

        public BusinessAwareRecoverer(KafkaTemplate<String, Object> kafkaTemplate) {
            this.kafkaTemplate = kafkaTemplate;
        }

        @Override
        public void accept(ConsumerRecord<?, ?> record, Exception exception) {
            log.error("Business-aware recovery for record: topic={}, key={}, error=
{}",
                record.topic(), record.key(), exception.getMessage());

            try {
                Object value = record.value();

                // Business-specific recovery logic
                if (value instanceof OrderEvent orderEvent) {
                    handleOrderEventFailure(orderEvent, exception);
                } else if (value instanceof PaymentEvent paymentEvent) {
                    handlePaymentEventFailure(paymentEvent, exception);
                } else {
                    handleGenericFailure(record, exception);
                }

            } catch (Exception e) {
                log.error("Recovery handling failed", e);
                // Fallback to standard DLT
                sendToStandardDLT(record, exception);
            }
        }

        private void handleOrderEventFailure(OrderEvent order, Exception ex) {
            log.info("Handling order event failure: orderId={}",
order.getOrderId());

            if (ex instanceof ValidationException) {
                // Send to validation failure topic
                kafkaTemplate.send("order-validation-failures", order.getOrderId(),
                    OrderValidationFailure.builder()
                        .originalOrder(order)
                        .error(ex.getMessage())
                        .timestamp(Instant.now())
                        .build());
            } else {
                // Send to general order failures
                kafkaTemplate.send("order-failures", order.getOrderId(), order);
            }
        }

        private void handlePaymentEventFailure(PaymentEvent payment, Exception ex) {
            log.info("Handling payment event failure: paymentId={}",
payment.getPaymentId());

            // Send to payment failures with enriched context
            PaymentFailure failure = PaymentFailure.builder()
                .originalPayment(payment)
                .failureReason(ex.getMessage())
                .failureType(determineFailureType(ex))
                .timestamp(Instant.now())
                .build();

            kafkaTemplate.send("payment-failures", payment.getPaymentId(), failure);
        }

        private void handleGenericFailure(ConsumerRecord<?, ?> record, Exception ex)
{
            log.info("Handling generic failure: topic={}", record.topic());

            // Send to generic DLT
            sendToStandardDLT(record, ex);
        }

        private void sendToStandardDLT(ConsumerRecord<?, ?> record, Exception ex) {
            String dltTopic = record.topic() + ".DLT";
            kafkaTemplate.send(dltTopic, record.key(), record.value());
        }

        private String determineFailureType(Exception ex) {
            if (ex instanceof ValidationException) return "VALIDATION";
            if (ex instanceof BusinessException) return "BUSINESS";
            if (ex instanceof TimeoutException) return "TIMEOUT";
            return "UNKNOWN";
        }
    }

    // Metrics methods
    private void updateRetryMetrics(String topic, int attempt) {
        log.debug("Updating retry metrics: topic={}, attempt={}", topic, attempt);
        // Implementation would update Micrometer metrics
    }

    private void updateRecoveryMetrics(String topic) {
        log.debug("Updating recovery metrics: topic={}", topic);
        // Implementation would update Micrometer metrics
    }

    private void updateFailureMetrics(String topic) {
        log.debug("Updating failure metrics: topic={}", topic);
        // Implementation would update Micrometer metrics
    }

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

/**
 * Error handling examples and patterns
 */
@Component
@lombok.extern.slf4j.Slf4j
public class ErrorHandlingExamples {

    /**
     * Resilient listener with comprehensive error handling
     */
    @KafkaListener(
        topics = "resilient-processing",
        groupId = "resilient-processor",
        containerFactory = "resilientContainerFactory"
    )
    public void handleResilientProcessing(@Payload OrderEvent order) {
        log.info("Processing order with resilient error handling: {}",
order.getOrderId());

        try {
            // Simulate different types of errors
            simulateProcessingScenarios(order);
            // Normal processing
            processOrder(order);

        } catch (TemporaryBusinessException e) {
            log.warn("Temporary business exception (will retry): {}",
e.getMessage());
            throw e; // Will be retried
        } catch (PermanentBusinessException e) {
            log.error("Permanent business exception (won't retry): {}",
e.getMessage());
            throw e; // Won't be retried
        } catch (Exception e) {
            log.error("Unexpected exception: {}", e.getMessage(), e);
            throw e; // Will use default retry logic
        }
    }

    /**
     * Fixed retry pattern listener
     */
    @KafkaListener(
        topics = "fixed-retry-processing",
        groupId = "fixed-retry-processor",
        containerFactory = "fixedRetryContainerFactory"
    )
    public void handleFixedRetryProcessing(@Payload PaymentEvent payment) {
        log.info("Processing payment with fixed retry: {}", payment.getPaymentId());

        try {
            processPayment(payment);
        } catch (ExternalServiceException e) {
            log.warn("External service error (will retry with fixed backoff): {}",
e.getMessage());
            throw e;
        }
    }

    /**
     * Business logic-aware error handling
     */
    @KafkaListener(
        topics = "business-logic-processing",
        groupId = "business-logic-processor",
        containerFactory = "businessLogicContainerFactory"
    )
    public void handleBusinessLogicProcessing(@Payload EventMessage event) {
        log.info("Processing event with business logic error handling: {}",
event.getEventId());

        try {
            validateBusinessRules(event);
            processBusinessEvent(event);

        } catch (DataValidationException e) {
            log.error("Data validation failed (won't retry): {}", e.getMessage());
```

```
                throw e; // Won't be retried, goes to validation DLT
            } catch (ExternalServiceException e) {
                log.warn("External service error (will retry): {}", e.getMessage());
                throw e; // Will be retried
            }
        }

        // Simulation and processing methods
        private void simulateProcessingScenarios(OrderEvent order) {
            String orderId = order.getOrderId();

            if (orderId.endsWith("TEMP_ERROR")) {
                throw new TemporaryBusinessException("Simulated temporary error");
            } else if (orderId.endsWith("PERM_ERROR")) {
                throw new PermanentBusinessException("Simulated permanent error");
            } else if (orderId.endsWith("TIMEOUT")) {
                throw new ExternalServiceException("Simulated timeout");
            }
            // Otherwise, proceed normally
        }

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

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

            // Simulate occasional external service failures
            if (Math.random() < 0.3) { // 30% failure rate
                throw new ExternalServiceException("Payment service temporarily
    unavailable");
            }
        }

        private void validateBusinessRules(EventMessage event) {
            if (event.getPayload() == null || event.getPayload().isEmpty()) {
                throw new DataValidationException("Event payload cannot be empty");
            }
        }

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

            // Simulate external service calls
            if (event.getEventType().equals("EXTERNAL_CALL") && Math.random() < 0.2) {
                throw new ExternalServiceException("External service call failed");
            }
        }
    }

    // Custom exception classes
    class TemporaryBusinessException extends RuntimeException {
        public TemporaryBusinessException(String message) { super(message); }
```

```
}

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

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

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

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

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

// Domain objects for error handling
@lombok.Data
@lombok.Builder
@lombok.NoArgsConstructor
@lombok.AllArgsConstructor
class OrderValidationFailure {
    private OrderEvent originalOrder;
    private String error;
    private Instant timestamp;
}

@lombok.Data
@lombok.Builder
@lombok.NoArgsConstructor
@lombok.AllArgsConstructor
class PaymentFailure {
    private PaymentEvent originalPayment;
    private String failureReason;
    private String failureType;
    private Instant timestamp;
}
```

📊 Comparisons & Trade-offs

Consumer Pattern Comparison

Pattern	Throughput	Latency	Reliability	Complexity	Use Case
Single Consumer	Low	Low	Medium	Low	Simple processing, ordering required

Pattern	Throughput	Latency	Reliability	Complexity	Use Case
Concurrent Consumers	High	Low	High	Medium	High-throughput parallel processing
Batch Processing	Highest	High	Medium	Medium	Bulk operations, high efficiency
Manual Ack	Medium	Medium	Highest	High	Critical processing, precise control
Auto Ack	Highest	Lowest	Lowest	Low	Development, non-critical processing

Error Handling Strategy Comparison

Strategy	Recovery Time	Data Loss Risk	Complexity	Monitoring	Use Case
No Retry	Immediate	High	Low	Simple	Non-critical, fire-and-forget
Fixed Retry	Predictable	Low	Medium	Medium	Stable retry requirements
Exponential Backoff	Variable	Low	Medium	Medium	Transient failures
Dead Letter Topic	N/A	None	High	Complex	Poison pill handling
Business Recovery	Contextual	Minimal	High	Complex	Business-critical processing

Acknowledgment Mode Trade-offs

Aspect	AUTO	MANUAL_IMMEDIATE	MANUAL	BATCH	RECORD
Performance	★★★★★	★★★	★★★★	★★★★★	★★
Reliability	★	★★★★★	★★★★	★★★	★★★★★
Complexity	★	★★★	★★★	★★★★	★★
Message Loss Risk	High	None	Low	Medium	None
Duplicate Risk	Low	Medium	Medium	Medium	High

🚨 Common Pitfalls & Best Practices

Common Anti-Patterns

❌ Configuration Mistakes

```
// DON'T - Mismatched consumer group and partition assignment
@KafkaListener(
    topics = "orders",
    groupId = "different-group-each-time-" + UUID.randomUUID()
) // Creates new group every restart!
public void badGroupIdPattern(OrderEvent order) {
    // This prevents proper consumer group management
}

// DON'T - Blocking operations in high-throughput listeners
@KafkaListener(topics = "high-volume-events", concurrency = "8")
public void badBlockingListener(EventMessage event) {
    // BLOCKING operation in high-concurrency listener
    String response = restTemplate.getForObject("http://slow-service",
String.class);
    processEvent(event, response); // Blocks all 8 threads!
}

// DON'T - Ignoring poison pills
@KafkaListener(topics = "unreliable-data")
public void badDeserializationHandling(OrderEvent order) {
    // No ErrorHandlingDeserializer configured
    // Poison pills will crash the consumer permanently
    processOrder(order);
}
```

❌ Error Handling Mistakes

```
// DON'T - Catching and ignoring exceptions
@KafkaListener(topics = "critical-orders")
public void badErrorHandling(OrderEvent order) {
    try {
        processCriticalOrder(order);
    } catch (Exception e) {
        // BAD: Silently ignoring errors
        log.error("Error processing order", e);
        // Message is acked but error is not handled!
    }
}

// DON'T - Infinite retry without circuit breaker
@KafkaListener(topics = "problematic-service")
public void badRetryPattern(ServiceRequest request) {
    // No circuit breaker - will retry forever
    callUnreliableService(request); // Keeps failing
}
```

Production Best Practices

☑ Optimal Configuration Patterns

```
/**
 * ☑ GOOD - Production-ready consumer configuration
 */
@Configuration
public class ProductionConsumerConfiguration {

    @Bean
    public ConsumerFactory<String, Object> productionConsumerFactory() {
        Map<String, Object> props = new HashMap<>();

        // Cluster configuration
        props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG,
            "kafka1:9092,kafka2:9092,kafka3:9092");

        // Consumer group stability
        props.put(ConsumerConfig.GROUP_ID_CONFIG, "production-consumer-v1");
        props.put(ConsumerConfig.CLIENT_ID_CONFIG,
            "consumer-" + InetAddress.getLocalHost().getHostName());

        // Performance optimization
        props.put(ConsumerConfig.MAX_POLL_RECORDS_CONFIG, 500);
        props.put(ConsumerConfig.MAX_POLL_INTERVAL_MS_CONFIG, 300000); // 5 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);
        props.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest");

        // Resilient deserializers
        props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
ErrorHandlingDeserializer.class);
        props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
ErrorHandlingDeserializer.class);
        props.put(ErrorHandlingDeserializer.KEY_DESERIALIZER_CLASS,
StringDeserializer.class);
        props.put(ErrorHandlingDeserializer.VALUE_DESERIALIZER_CLASS,
JsonDeserializer.class);

        return new DefaultKafkaConsumerFactory<>(props);
    }

    /**
     * ☑ GOOD - Production container factory with comprehensive error handling
     */
    @Bean
    public ConcurrentKafkaListenerContainerFactory<String, Object>
productionContainerFactory() {
        ConcurrentKafkaListenerContainerFactory<String, Object> factory =
            new ConcurrentKafkaListenerContainerFactory<>();

        factory.setConsumerFactory(productionConsumerFactory());

        // Optimal concurrency
        factory.setConcurrency(Runtime.getRuntime().availableProcessors());

        // Manual acknowledgment for reliability

factory.getContainerProperties().setAckMode(ContainerProperties.AckMode.MANUAL_IMMED
IATE);

        // Comprehensive error handling
        factory.setCommonErrorHandler(productionErrorHandler());

        // Rebalance listener for monitoring

factory.getContainerProperties().setConsumerRebalanceListener(rebalanceMonitor());

        return factory;
    }

    @Bean
    public CommonErrorHandler productionErrorHandler() {
        // Business-aware dead letter routing
        DeadLetterPublishingRecoverer recoverer = new DeadLetterPublishingRecoverer(
            kafkaTemplate(),
            (record, ex) -> {
                String dltSuffix = determineDltSuffix(ex);
                return new TopicPartition(record.topic() + dltSuffix,
record.partition());
            }
        );

        // Exponential backoff with reasonable limits
        ExponentialBackOff backOff = new ExponentialBackOff(1000L, 2.0);
        backOff.setMaxElapsedTime(60000L); // Max 1 minute total retry time

        DefaultErrorHandler errorHandler = new DefaultErrorHandler(recoverer,
backOff);

        // Classify exceptions appropriately
        errorHandler.addNotRetryableExceptions(
            IllegalArgumentException.class,
            ValidationException.class,
            DeserializationException.class
        );

        // Add comprehensive monitoring
        errorHandler.setRetryListeners(productionRetryListener());

        return errorHandler;
    }

    private String determineDltSuffix(Exception ex) {
        if (ex instanceof ValidationException) return ".validation.DLT";
        if (ex instanceof BusinessException) return ".business.DLT";
        if (ex instanceof TimeoutException) return ".timeout.DLT";
        return ".DLT";
    }
```

```
}

/**
 * ☑ GOOD - Production service patterns
 */
@Service
@lombok.extern.slf4j.Slf4j
public class ProductionConsumerService {

    private final CircuitBreaker circuitBreaker =
CircuitBreaker.ofDefaults("external-service");
    private final MeterRegistry meterRegistry = Metrics.globalRegistry;

    /**
     * ☑ GOOD - Comprehensive message processing with all best practices
     */
    @KafkaListener(
        topics = "production-orders",
        groupId = "order-processor-v1",
        containerFactory = "productionContainerFactory"
    )
    @Timed(name = "order.processing.time", description = "Order processing time")
    public void processProductionOrder(@Payload OrderEvent order,
                                       @Header(KafkaHeaders.RECEIVED_TOPIC) String
topic,
                                       @Header(KafkaHeaders.RECEIVED_PARTITION_ID) int
partition,
                                       @Header(KafkaHeaders.OFFSET) long offset,
                                       Acknowledgment ack) {

        String orderId = order.getOrderId();
        Timer.Sample sample = Timer.start(meterRegistry);

        log.info("Processing order: id={}, topic={}, partition={}, offset={}",
            orderId, topic, partition, offset);

        try {
            // 1. Validate input
            validateOrder(order);

            // 2. Check for duplicates (idempotency)
            if (isDuplicateOrder(orderId)) {
                log.info("Duplicate order detected, skipping: {}", orderId);
                ack.acknowledge();
                return;
            }

            // 3. Process with circuit breaker
            Supplier<Void> processSupplier =
CircuitBreaker.decorateSupplier(circuitBreaker, () -> {
                processOrderBusinessLogic(order);
                return null;
            });

            processSupplier.get();
            // 4. Mark as processed (idempotency tracking)
            markOrderAsProcessed(orderId);

            // 5. Acknowledge only after successful processing
            ack.acknowledge();

            // 6. Update success metrics
            meterRegistry.counter("order.processing.success", "topic",
topic).increment();

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

        } catch (ValidationException e) {
            log.error("Order validation failed: orderId={}, error={}", orderId,
e.getMessage());
            // Acknowledge to avoid retries for invalid data
            ack.acknowledge();
            meterRegistry.counter("order.processing.validation_error", "topic",
topic).increment();

        } catch (DuplicateOrderException e) {
            log.warn("Duplicate order processing attempt: {}", orderId);
            // Acknowledge as it's already processed
            ack.acknowledge();

        } catch (Exception e) {
            log.error("Order processing failed: orderId={}", orderId, e);
            meterRegistry.counter("order.processing.error", "topic",
topic).increment();
            // Don't acknowledge - will be retried
            throw e;

        } finally {
            sample.stop(Timer.builder("order.processing.duration")
                .tag("topic", topic)
                .register(meterRegistry));
        }
    }

    /**
     * ☑ GOOD - Async processing with proper resource management
     */
    @KafkaListener(
        topics = "async-processing",
        groupId = "async-processor",
        concurrency = "8"
    )
    public CompletableFuture<Void> processAsynchronously(
            @Payload ProcessingTask task,
            Acknowledgment ack) {

        return CompletableFuture
            .supplyAsync(() -> {
                processTask(task);
                return null;
            })
            .whenComplete((result, ex) -> {
                if (ex != null) {
                    log.error("Async processing failed: taskId={}",
task.getTaskId(), ex);
                    // Error will be handled by error handler
                } else {
                    log.info("Async processing completed: taskId={}",
task.getTaskId());
                    ack.acknowledge();
                }
            });
    }

    /**
     * ✅ GOOD - Batch processing with partial failure handling
     */
    @KafkaListener(
        topics = "batch-processing",
        groupId = "batch-processor",
        containerFactory = "batchContainerFactory"
    )
    public void processBatchWithPartialFailureHandling(
            List<BatchItem> items,
            List<ConsumerRecord<String, BatchItem>> records,
            Acknowledgment ack) {

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

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

        // Process items individually to handle partial failures
        for (int i = 0; i < items.size(); i++) {
            try {
                BatchItem item = items.get(i);
                processBatchItem(item);
                successful.add(item);

            } catch (Exception e) {
                log.error("Failed to process batch item at index {}: {}", i,
e.getMessage());
                failed.add(items.get(i));

                // Send failed item to retry topic
                sendToRetryTopic(items.get(i), e);
            }
        }

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

        // Acknowledge batch even with partial failures
        // Failed items are sent to retry topic
        ack.acknowledge();

        // Update metrics

meterRegistry.counter("batch.processing.items.success").increment(successful.size())
;

meterRegistry.counter("batch.processing.items.failed").increment(failed.size());
    }

    // Helper methods implementing best practices
    private void validateOrder(OrderEvent order) throws ValidationException {
        if (order.getOrderId() == null || order.getOrderId().isEmpty()) {
            throw new ValidationException("Order ID cannot be null or empty");
        }
        if (order.getAmount() == null ||
order.getAmount().compareTo(BigDecimal.ZERO) <= 0) {
            throw new ValidationException("Order amount must be positive");
        }
    }

    private boolean isDuplicateOrder(String orderId) {
        // Check Redis, database, or in-memory cache
        return duplicateTracker.contains(orderId);
    }

    private void markOrderAsProcessed(String orderId) {
        // Store in Redis, database, or in-memory cache
        duplicateTracker.add(orderId);
    }

    private void processOrderBusinessLogic(OrderEvent order) {
        // Actual business logic here
        log.debug("Processing order business logic: {}", order.getOrderId());

        // Simulate processing time
        try {
            Thread.sleep(100);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
            throw new RuntimeException("Processing interrupted", e);
        }
    }

    private void processTask(ProcessingTask task) {
        log.debug("Processing task: {}", task.getTaskId());
        // Task processing logic
    }

    private void processBatchItem(BatchItem item) {
        log.debug("Processing batch item: {}", item.getItemId());

        // Simulate occasional failures
        if (item.getItemId().endsWith("FAIL")) {
            throw new RuntimeException("Simulated batch item failure");
        }
    }

    private void sendToRetryTopic(BatchItem item, Exception e) {
        log.info("Sending failed batch item to retry topic: {}", item.getItemId());

        RetryableItem retryItem = RetryableItem.builder()
            .originalItem(item)
            .error(e.getMessage())
            .retryCount(1)
            .timestamp(Instant.now())
            .build();

        kafkaTemplate.send("batch-processing.retry", item.getItemId(), retryItem);
    }

    // Injected dependencies
    @Autowired private DuplicateTracker duplicateTracker;
    @Autowired private KafkaTemplate<String, Object> kafkaTemplate;
}

// Supporting classes
interface DuplicateTracker {
    boolean contains(String id);
    void add(String id);
}

@lombok.Data
@lombok.Builder
@lombok.NoArgsConstructor
@lombok.AllArgsConstructor
class BatchItem {
    private String itemId;
    private String data;
    private Instant timestamp;
}

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

@lombok.Data
@lombok.Builder
@lombok.NoArgsConstructor
@lombok.AllArgsConstructor
class RetryableItem {
    private BatchItem originalItem;
    private String error;
    private int retryCount;
    private Instant timestamp;
}

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

☑ Operational Excellence Patterns

```
/**
 * ☑ GOOD - Health monitoring and observability
 */
@Component
public class ConsumerHealthMonitoring {

    @Autowired
    private KafkaListenerEndpointRegistry registry;

    @Autowired
    private MeterRegistry meterRegistry;

    /**
     * Health check endpoint
     */
    @EventListener
    @Scheduled(fixedDelay = 30000) // Every 30 seconds
    public void monitorConsumerHealth() {
        Collection<MessageListenerContainer> containers =
registry.getAllListenerContainers();

        for (MessageListenerContainer container : containers) {
            String listenerId = container.getListenerId();
            boolean isRunning = container.isRunning();

            // Update health metrics
            meterRegistry.gauge("kafka.consumer.health",
                Tags.of("listener.id", listenerId),
                isRunning ? 1.0 : 0.0);

            if (!isRunning) {
                log.error("🚨 Consumer container is not running: {}", listenerId);
                alertOperationsTeam(listenerId);
            }
        }
    }

    /**
     * Graceful shutdown handling
     */
    @PreDestroy
    public void shutdown() {
        log.info("Initiating graceful consumer shutdown");

        Collection<MessageListenerContainer> containers = 
registry.getAllListenerContainers();

        // Stop all containers gracefully
        for (MessageListenerContainer container : containers) {
            String listenerId = container.getListenerId();
            log.info("Stopping consumer container: {}", listenerId);

            container.stop();

            // Wait for graceful shutdown
            try {
                Thread.sleep(5000); // 5 second grace period
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
                log.warn("Shutdown interrupted for container: {}", listenerId);
            }
        }

        log.info("Consumer shutdown completed");
    }

    private void alertOperationsTeam(String listenerId) {
        log.error("Alerting operations team about container failure: {}", 
listenerId);
        // Integration with PagerDuty, Slack, etc.
    }
}
```

🌍 Real-World Use Cases

E-commerce Order Processing Pipeline

```
/**
 * Production e-commerce order processing with Spring Kafka consumers
 */
@Service
public class EcommerceOrderConsumerService {

    /**
     * Order placement consumer with validation and inventory check
     */
    @KafkaListener(
        topics = "order-placed",
        groupId = "order-processing-service",
        containerFactory = "resilientContainerFactory"
    )
    public void processOrderPlaced(@Payload OrderPlacedEvent orderEvent,
                                  Acknowledgment ack) {
```

```
        log.info("Processing order placement: orderId={}, customerId={}, amount={}",
            orderEvent.getOrderId(), orderEvent.getCustomerId(),
orderEvent.getTotalAmount());

        try {
            // 1. Validate order
            validateOrderPlacement(orderEvent);

            // 2. Check inventory
            InventoryCheckResult inventoryResult =
checkInventoryAvailability(orderEvent);

            if (inventoryResult.isAvailable()) {
                // 3. Reserve inventory
                reserveInventory(orderEvent);

                // 4. Calculate pricing
                PricingResult pricing = calculatePricing(orderEvent);

                // 5. Process payment
                initiatePaymentProcessing(orderEvent, pricing);

                // 6. Update order status
                updateOrderStatus(orderEvent.getOrderId(), "CONFIRMED");

            } else {
                // Handle out of stock
                handleOutOfStock(orderEvent, inventoryResult);
            }

            ack.acknowledge();

        } catch (ValidationException e) {
            log.error("Order validation failed: {}", e.getMessage());
            handleOrderValidationFailure(orderEvent, e);
            ack.acknowledge(); // Don't retry validation failures

        } catch (Exception e) {
            log.error("Order processing failed: orderId={}",
orderEvent.getOrderId(), e);
            throw e; // Will be retried
        }
    }

    /**
     * Payment processing result consumer
     */
    @KafkaListener(
        topics = "payment-processed",
        groupId = "order-payment-service"
    )
    public void processPaymentResult(@Payload PaymentProcessedEvent paymentEvent,
                                     Acknowledgment ack) {

        try {
            if (paymentEvent.isSuccessful()) {
                // Payment successful - proceed with fulfillment
                initiateFulfillment(paymentEvent.getOrderId());
                updateOrderStatus(paymentEvent.getOrderId(), "PAID");

            } else {
                // Payment failed - cancel order
                cancelOrder(paymentEvent.getOrderId(),
paymentEvent.getFailureReason());
                releaseInventory(paymentEvent.getOrderId());
            }

            ack.acknowledge();

        } catch (Exception e) {
            log.error("Payment result processing failed: orderId={}",
                paymentEvent.getOrderId(), e);
            throw e;
        }
    }

    /**
     * Batch inventory updates consumer
     */
    @KafkaListener(
        topics = "inventory-updates",
        groupId = "inventory-processor",
        containerFactory = "batchContainerFactory"
    )
    public void processInventoryUpdates(List<InventoryUpdateEvent> updates,
                                        Acknowledgment ack) {

        log.info("Processing batch inventory updates: {} items", updates.size());

        try {
            // Group updates by product for efficiency
            Map<String, List<InventoryUpdateEvent>> updatesByProduct =
updates.stream()
                .collect(Collectors.groupingBy(InventoryUpdateEvent::getProductId));

            // Process each product's updates
            updatesByProduct.forEach(this::processProductInventoryUpdates);

            ack.acknowledge();

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

    // Business logic implementations
    private void validateOrderPlacement(OrderPlacedEvent order) throws
ValidationException {
        if (order.getItems().isEmpty()) {
            throw new ValidationException("Order must contain at least one item");
        }

        for (OrderItem item : order.getItems()) {
            if (item.getQuantity() <= 0) {
                throw new ValidationException("Item quantity must be positive");
            }
        }
    }

    private InventoryCheckResult checkInventoryAvailability(OrderPlacedEvent order)
{
        // Check inventory for all items
        return inventoryService.checkAvailability(order.getItems());
    }

    private void reserveInventory(OrderPlacedEvent order) {
        inventoryService.reserveItems(order.getOrderId(), order.getItems());
    }

    private PricingResult calculatePricing(OrderPlacedEvent order) {
        return pricingService.calculateTotalPrice(order);
    }

    private void initiatePaymentProcessing(OrderPlacedEvent order, PricingResult
pricing) {
        PaymentRequest paymentRequest = PaymentRequest.builder()
            .orderId(order.getOrderId())
            .customerId(order.getCustomerId())
            .amount(pricing.getTotalAmount())
            .paymentMethod(order.getPaymentMethod())
            .build();

        paymentService.processPayment(paymentRequest);
    }

    private void updateOrderStatus(String orderId, String status) {
        orderService.updateStatus(orderId, status);
    }

    private void handleOutOfStock(OrderPlacedEvent order, InventoryCheckResult
result) {
        log.warn("Order has out of stock items: orderId={}, unavailableItems={}",
            order.getOrderId(), result.getUnavailableItems());

        // Send out of stock notification
        OutOfStockEvent outOfStockEvent = OutOfStockEvent.builder()
            .orderId(order.getOrderId())
            .customerId(order.getCustomerId())
            .unavailableItems(result.getUnavailableItems())
            .build();

        eventPublisher.publishEvent("order-out-of-stock", outOfStockEvent);
    }
    private void handleOrderValidationFailure(OrderPlacedEvent order,
ValidationException e) {
        ValidationFailureEvent failureEvent = ValidationFailureEvent.builder()
            .orderId(order.getOrderId())
            .customerId(order.getCustomerId())
            .validationErrors(List.of(e.getMessage()))
            .build();

        eventPublisher.publishEvent("order-validation-failed", failureEvent);
    }

    private void initiateFulfillment(String orderId) {
        fulfillmentService.createFulfillmentOrder(orderId);
    }

    private void cancelOrder(String orderId, String reason) {
        orderService.cancelOrder(orderId, reason);
    }

    private void releaseInventory(String orderId) {
        inventoryService.releaseReservation(orderId);
    }

    private void processProductInventoryUpdates(String productId,
                                                List<InventoryUpdateEvent> updates) {
        log.debug("Processing {} inventory updates for product: {}",
            updates.size(), productId);

        inventoryService.applyBatchUpdates(productId, updates);
    }

    // Injected services
    @Autowired private InventoryService inventoryService;
    @Autowired private PricingService pricingService;
    @Autowired private PaymentService paymentService;
    @Autowired private OrderService orderService;
    @Autowired private FulfillmentService fulfillmentService;
    @Autowired private EventPublisher eventPublisher;
}
```

Financial Transaction Processing

```
/**
 * Financial transaction processing with exactly-once guarantees
 */
@Service
public class FinancialTransactionConsumerService {

    /**
     * Money transfer processing with idempotency
     */
    @KafkaListener(
        topics = "money-transfer-requests",
        groupId = "financial-processor",
        containerFactory = "exactlyOnceContainerFactory"
    )
    @Transactional
    public void processMoneyTransfer(@Payload MoneyTransferRequest transfer,
                                     Acknowledgment ack) {

        String transactionId = transfer.getTransactionId();

        log.info("Processing money transfer: transactionId={}, amount={}, from={},
to={}",
            transactionId, transfer.getAmount(),
            transfer.getFromAccount(), transfer.getToAccount());

        try {
            // 1. Idempotency check
            if (isTransactionAlreadyProcessed(transactionId)) {
                log.info("Transaction already processed: {}", transactionId);
                ack.acknowledge();
                return;
            }

            // 2. Validate accounts and amount
            validateTransfer(transfer);

            // 3. Check account balances
            if (!hasSufficientBalance(transfer.getFromAccount(),
transfer.getAmount())) {
                handleInsufficientFunds(transfer);
                ack.acknowledge();
                return;
            }

            // 4. Execute transfer atomically
            executeAtomicTransfer(transfer);

            // 5. Mark transaction as processed
            markTransactionAsProcessed(transactionId);

            // 6. Send confirmation
            sendTransferConfirmation(transfer);

            ack.acknowledge();

            log.info("Money transfer completed successfully: {}", transactionId);

        } catch (ValidationException e) {
            log.error("Transfer validation failed: {}", e.getMessage());
            handleValidationFailure(transfer, e);
            ack.acknowledge(); // Don't retry validation failures

        } catch (Exception e) {
            log.error("Transfer processing failed: transactionId={}", transactionId,
e);
            throw e; // Will be retried
        }
    }

    /**
     * Account balance update consumer
     */
    @KafkaListener(
        topics = "account-balance-updates",
        groupId = "balance-processor"
    )
    public void processBalanceUpdate(@Payload AccountBalanceUpdateEvent updateEvent,
                                     Acknowledgment ack) {

        try {
            // Apply balance update with optimistic locking
            accountService.updateBalance(
                updateEvent.getAccountId(),
                updateEvent.getAmount(),
                updateEvent.getTransactionId()
            );

            // Trigger balance alerts if necessary
            checkBalanceThresholds(updateEvent.getAccountId());

            ack.acknowledge();

        } catch (OptimisticLockingFailureException e) {
            log.warn("Optimistic locking failed for balance update, retrying:
accountId={}",
                updateEvent.getAccountId());
            throw e; // Will be retried

        } catch (Exception e) {
            log.error("Balance update failed: accountId={}",
updateEvent.getAccountId(), e);
            throw e;
        }
    }

    // Financial business logic
    private boolean isTransactionAlreadyProcessed(String transactionId) {
        return transactionRepository.existsByTransactionId(transactionId);
    }

    private void validateTransfer(MoneyTransferRequest transfer) throws
ValidationException {
        if (transfer.getAmount().compareTo(BigDecimal.ZERO) <= 0) {
            throw new ValidationException("Transfer amount must be positive");
        }

        if (transfer.getFromAccount().equals(transfer.getToAccount())) {
            throw new ValidationException("Cannot transfer to the same account");
        }

        if (!accountService.accountExists(transfer.getFromAccount()) ||
            !accountService.accountExists(transfer.getToAccount())) {
            throw new ValidationException("One or both accounts do not exist");
        }
    }

    private boolean hasSufficientBalance(String accountId, BigDecimal amount) {
        BigDecimal currentBalance = accountService.getBalance(accountId);
        return currentBalance.compareTo(amount) >= 0;
    }

    private void executeAtomicTransfer(MoneyTransferRequest transfer) {
        // Debit from source account
        accountService.debitAccount(transfer.getFromAccount(),
transfer.getAmount());

        // Credit to destination account
        accountService.creditAccount(transfer.getToAccount(), transfer.getAmount());

        // Record transaction
        recordTransaction(transfer);
    }

    private void recordTransaction(MoneyTransferRequest transfer) {
        TransactionRecord record = TransactionRecord.builder()
            .transactionId(transfer.getTransactionId())
            .fromAccount(transfer.getFromAccount())
            .toAccount(transfer.getToAccount())
            .amount(transfer.getAmount())
            .status("COMPLETED")
            .timestamp(Instant.now())
            .build();

        transactionRepository.save(record);
    }

    private void markTransactionAsProcessed(String transactionId) {
        // Mark in idempotency store
        idempotencyService.markAsProcessed(transactionId);
    }

    private void sendTransferConfirmation(MoneyTransferRequest transfer) {
        TransferConfirmationEvent confirmation = TransferConfirmationEvent.builder()
            .transactionId(transfer.getTransactionId())
            .fromAccount(transfer.getFromAccount())
            .toAccount(transfer.getToAccount())
            .amount(transfer.getAmount())
            .status("COMPLETED")
            .timestamp(Instant.now())
            .build();

        eventPublisher.publishEvent("transfer-confirmations", confirmation);
    }

    private void handleInsufficientFunds(MoneyTransferRequest transfer) {
        InsufficientFundsEvent event = InsufficientFundsEvent.builder()
```

```
            .transactionId(transfer.getTransactionId())
            .accountId(transfer.getFromAccount())
            .requestedAmount(transfer.getAmount())
            .availableBalance(accountService.getBalance(transfer.getFromAccount()))
            .build();

        eventPublisher.publishEvent("insufficient-funds", event);
    }

    private void handleValidationFailure(MoneyTransferRequest transfer,
ValidationException e) {
        ValidationFailureEvent failure = ValidationFailureEvent.builder()
            .transactionId(transfer.getTransactionId())
            .error(e.getMessage())
            .timestamp(Instant.now())
            .build();

        eventPublisher.publishEvent("transfer-validation-failures", failure);
    }

    private void checkBalanceThresholds(String accountId) {
        BigDecimal balance = accountService.getBalance(accountId);
        BigDecimal threshold = accountService.getLowBalanceThreshold(accountId);

        if (balance.compareTo(threshold) <= 0) {
            LowBalanceAlertEvent alert = LowBalanceAlertEvent.builder()
                .accountId(accountId)
                .currentBalance(balance)
                .threshold(threshold)
                .build();

            eventPublisher.publishEvent("low-balance-alerts", alert);
        }
    }

    // Injected services
    @Autowired private AccountService accountService;
    @Autowired private TransactionRepository transactionRepository;
    @Autowired private IdempotencyService idempotencyService;
    @Autowired private EventPublisher eventPublisher;
}
```

IoT Data Processing Pipeline

```
/**
 * IoT sensor data processing with time-window aggregation
 */
@Service
public class IoTDataConsumerService {

    private final Map<String, SensorDataAggregator> aggregators = new
ConcurrentHashMap<>();
```

```
    /**
     * Real-time sensor data consumer with aggregation
     */
    @KafkaListener(
        topics = "sensor-data",
        groupId = "iot-data-processor",
        concurrency = "6"
    )
    public void processSensorData(@Payload SensorDataEvent sensorData,
                                  @Header(KafkaHeaders.RECEIVED_PARTITION_ID) int
partition,
                                  Acknowledgment ack) {

        String sensorId = sensorData.getSensorId();

        try {
            // 1. Validate sensor data
            validateSensorData(sensorData);

            // 2. Get or create aggregator for sensor
            SensorDataAggregator aggregator = aggregators.computeIfAbsent(
                sensorId, k -> new SensorDataAggregator(sensorId));

            // 3. Add data point to aggregator
            aggregator.addDataPoint(sensorData);

            // 4. Check if window is complete
            if (aggregator.isWindowComplete()) {
                SensorAggregateEvent aggregate = aggregator.getAggregate();
                publishSensorAggregate(aggregate);
                aggregator.reset();
            }

            // 5. Store raw data for historical analysis
            storeRawSensorData(sensorData);

            // 6. Check for anomalies
            checkForAnomalies(sensorData);

            ack.acknowledge();

        } catch (ValidationException e) {
            log.error("Invalid sensor data: sensorId={}, error={}", sensorId,
e.getMessage());
            ack.acknowledge(); // Don't retry invalid data

        } catch (Exception e) {
            log.error("Sensor data processing failed: sensorId={}", sensorId, e);
            throw e;
        }
    }

    /**
     * Device status updates consumer
     */
    @KafkaListener(
        topics = "device-status",
        groupId = "device-monitor"
    )
    public void processDeviceStatus(@Payload DeviceStatusEvent statusEvent,
                                    Acknowledgment ack) {

        try {
            String deviceId = statusEvent.getDeviceId();

            // Update device registry
            deviceRegistry.updateStatus(deviceId, statusEvent.getStatus());

            // Check for device health issues
            if ("OFFLINE".equals(statusEvent.getStatus()) ||
"ERROR".equals(statusEvent.getStatus())) {
                handleDeviceHealthIssue(statusEvent);
            }

            // Update monitoring dashboards
            updateDeviceMetrics(statusEvent);

            ack.acknowledge();

        } catch (Exception e) {
            log.error("Device status processing failed: deviceId={}",
                statusEvent.getDeviceId(), e);
            throw e;
        }
    }

    /**
     * Batch processing for historical data analysis
     */
    @KafkaListener(
        topics = "sensor-data-batch",
        groupId = "historical-analyzer",
        containerFactory = "batchContainerFactory"
    )
    public void processHistoricalData(List<SensorDataEvent> sensorDataBatch,
                                      Acknowledgment ack) {

        log.info("Processing historical sensor data batch: {} records",
sensorDataBatch.size());

        try {
            // Group by sensor for efficient processing
            Map<String, List<SensorDataEvent>> dataBySensor =
sensorDataBatch.stream()
                .collect(Collectors.groupingBy(SensorDataEvent::getSensorId));

            // Process each sensor's data
            dataBySensor.forEach(this::processHistoricalDataForSensor);

            ack.acknowledge();
        } catch (Exception e) {
            log.error("Historical data batch processing failed", e);
            throw e;
        }
    }

    // IoT business logic
    private void validateSensorData(SensorDataEvent data) throws ValidationException
{
        if (data.getValue() < data.getMinValue() || data.getValue() >
data.getMaxValue()) {
            throw new ValidationException("Sensor value out of valid range");
        }

        if (data.getTimestamp().isAfter(Instant.now().plusSeconds(60))) {
            throw new ValidationException("Sensor timestamp is in the future");
        }
    }

    private void publishSensorAggregate(SensorAggregateEvent aggregate) {
        eventPublisher.publishEvent("sensor-aggregates", aggregate);
    }

    private void storeRawSensorData(SensorDataEvent data) {
        // Store in time-series database
        timeSeriesDatabase.store(data);
    }

    private void checkForAnomalies(SensorDataEvent data) {
        if (anomalyDetector.isAnomaly(data)) {
            AnomalyDetectedEvent anomaly = AnomalyDetectedEvent.builder()
                .sensorId(data.getSensorId())
                .value(data.getValue())
                .expectedRange(anomalyDetector.getExpectedRange(data.getSensorId()))
                .severity(anomalyDetector.calculateSeverity(data))
                .timestamp(data.getTimestamp())
                .build();

            eventPublisher.publishEvent("sensor-anomalies", anomaly);

            // Alert if critical
            if (anomaly.getSeverity() == AnomalySeverity.CRITICAL) {
                alertService.sendCriticalAnomalyAlert(anomaly);
            }
        }
    }

    private void handleDeviceHealthIssue(DeviceStatusEvent statusEvent) {
        DeviceHealthIssueEvent healthIssue = DeviceHealthIssueEvent.builder()
            .deviceId(statusEvent.getDeviceId())
            .status(statusEvent.getStatus())
            .lastSeen(statusEvent.getTimestamp())
            .location(deviceRegistry.getDeviceLocation(statusEvent.getDeviceId()))
            .build();
        eventPublisher.publishEvent("device-health-issues", healthIssue);

        // Auto-recovery attempts
        if ("OFFLINE".equals(statusEvent.getStatus())) {
            deviceControlService.attemptReconnection(statusEvent.getDeviceId());
        }
    }

    private void updateDeviceMetrics(DeviceStatusEvent statusEvent) {
        meterRegistry.gauge("device.status",
            Tags.of("device.id", statusEvent.getDeviceId()),
            "ONLINE".equals(statusEvent.getStatus()) ? 1.0 : 0.0);
    }

    private void processHistoricalDataForSensor(String sensorId,
List<SensorDataEvent> data) {
        log.debug("Processing historical data for sensor {}: {} data points",
sensorId, data.size());

        // Perform statistical analysis
        SensorStatistics stats = statisticsCalculator.calculate(data);

        // Store aggregated statistics
        historicalStatsRepository.save(sensorId, stats);

        // Update ML models if needed
        if (data.size() >= ML_TRAINING_THRESHOLD) {
            machineLearningService.updateAnomalyModel(sensorId, data);
        }
    }

    // Injected services
    @Autowired private DeviceRegistry deviceRegistry;
    @Autowired private TimeSeriesDatabase timeSeriesDatabase;
    @Autowired private AnomalyDetector anomalyDetector;
    @Autowired private AlertService alertService;
    @Autowired private DeviceControlService deviceControlService;
    @Autowired private StatisticsCalculator statisticsCalculator;
    @Autowired private HistoricalStatsRepository historicalStatsRepository;
    @Autowired private MachineLearningService machineLearningService;
    @Autowired private EventPublisher eventPublisher;
    @Autowired private MeterRegistry meterRegistry;

    private static final int ML_TRAINING_THRESHOLD = 1000;
}
```

📈 Version Highlights

Spring Kafka Consumer Evolution Timeline

Version	Release	Key Consumer Features

Version	Release	Key Consumer Features
3.1.x	2024	Enhanced error handling, enforced rebalancing API, improved observability
3.0.x	2023	Spring Boot 3 support, GraalVM native compilation, performance improvements
2.9.x	2022	Non-blocking retries, enhanced DLT handling, better batch processing
2.8.x	2022	DefaultErrorHandler introduction, improved retry mechanisms
2.7.x	2021	Enhanced batch listeners, better error recovery strategies
2.6.x	2021	Non-blocking retries, RetryableTopic annotation, DLT enhancements
2.5.x	2020	ErrorHandlingDeserializer improvements, better security
2.4.x	2020	Pause/resume functionality, container scaling improvements
2.3.x	2019	ReplyingKafkaTemplate, request-reply patterns
2.2.x	2018	@KafkaListener enhancements, transaction support

Modern Consumer Features (2023-2025)

Spring Kafka 3.1+ Consumer Enhancements:

- **Enforced Rebalancing**: Programmatic control over consumer group rebalancing
- **Enhanced Error Handling**: More sophisticated retry and recovery mechanisms
- **Better Observability**: Improved metrics, tracing, and health monitoring
- **Performance Optimizations**: Better memory usage and throughput
- **Native Compilation**: GraalVM support for faster startup times

Key Configuration Evolution:

- **Simplified Error Handling**: Reduced boilerplate for error handler configuration
- **Better Default Configurations**: Production-ready defaults for most scenarios
- **Enhanced Deserialization**: Improved poison pill handling and error recovery
- **Improved Batch Processing**: Better batch listener performance and error handling

---

🔗 Additional Resources

📚 Official Documentation

- Spring Kafka Consumer Reference
- Apache Kafka Consumer Configuration
- Spring Kafka Error Handling

🎓 Learning Resources

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

🛠️ Development Tools

- Kafka Tool - GUI for Kafka cluster management
- Conduktor - Modern Kafka desktop client
- AKHQ - Web interface for managing Kafka clusters

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

> 💡 **Pro Tip**: Start with error-handling deserializers to handle poison pills, use manual acknowledgment for critical processing, implement proper rebalance listeners for production workloads, and always monitor consumer lag and health metrics. The combination of Spring's annotations with Kafka's robust consumer features provides excellent foundation for building reliable, scalable message-driven applications.