Spring Kafka Error Handling & Retry: Part 3 - Legacy Handlers, Best Practices & Production Patterns

Final part of the comprehensive guide covering SeekToCurrentErrorHandler migration, comparisons, best practices, and real-world use cases.

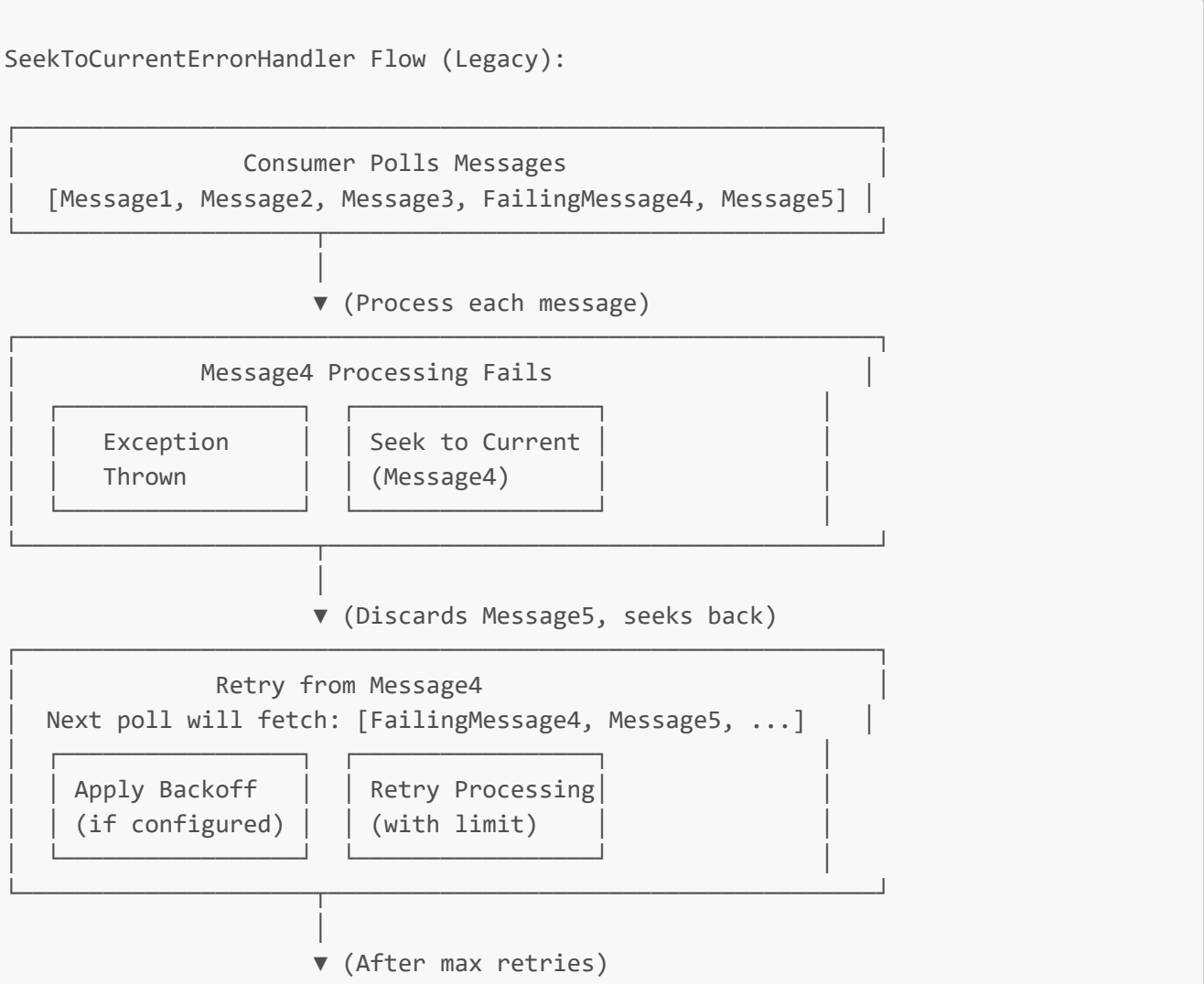
🔄 SeekToCurrentErrorHandler (Legacy)

Simple Explanation: SeekToCurrentErrorHandler was the default error handler in Spring Kafka before version 2.8. It works by seeking the consumer back to the current offset after a failure, allowing the same message to be retried. While deprecated, understanding it is important for legacy system maintenance and migration.

Why SeekToCurrentErrorHandler was Used:

- **Simple Retry Logic**: Straightforward approach to message retry
- **Partition Isolation**: Failures in one partition don't affect others
- **Backoff Support**: Configurable retry intervals
- **Recovery Integration**: Could be combined with recoverers for DLT

Legacy vs Modern Architecture:



```
┌─────────────────────────────────────────────────────────┐
│                    Recovery                             │
│  Send to DLT or Log (if recoverer configured)           │
└─────────────────────────────────────────────────────────┘
```

Legacy SeekToCurrentErrorHandler Implementation

```
import org.springframework.kafka.listener.SeekToCurrentErrorHandler;
import org.springframework.kafka.listener.DeadLetterPublishingRecoverer;
import org.springframework.util.backoff.FixedBackOff;
import org.springframework.util.backoff.ExponentialBackOff;

/**
 * Legacy SeekToCurrentErrorHandler configuration (deprecated)
 * This is for reference and migration purposes only
 */
@Configuration
@lombok.extern.slf4j.Slf4j
public class LegacyErrorHandlerConfiguration {

    @Autowired
    private KafkaTemplate<String, Object> kafkaTemplate;

    /**
     * Basic SeekToCurrentErrorHandler configuration (DEPRECATED)
     *
     * @deprecated Use DefaultErrorHandler instead
     */
    @Bean("legacySeekToCurrentErrorHandler")
    @Deprecated
    public SeekToCurrentErrorHandler legacySeekToCurrentErrorHandler() {

        // Simple recoverer that logs failed messages
        BiConsumer<ConsumerRecord<?, ?>, Exception> recoverer = (record,
exception) -> {
            log.error("Failed to process record after retries: topic={},
partition={}, offset={}, error={}",
                record.topic(), record.partition(), record.offset(),
exception.getMessage());
        };

        // Fixed backoff: 3 retries with 2 second intervals
        FixedBackOff backOff = new FixedBackOff(2000L, 3L);

        SeekToCurrentErrorHandler errorHandler = new
SeekToCurrentErrorHandler(recoverer, backOff);

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

```
org.springframework.kafka.support.serializer.DeserializationException.class
        );

        log.warn("Using deprecated SeekToCurrentErrorHandler - consider migrating
to DefaultErrorHandler");

        return errorHandler;
    }

    /**
     * SeekToCurrentErrorHandler with DLT recovery (DEPRECATED)
     *
     * @deprecated Use DefaultErrorHandler with DeadLetterPublishingRecoverer
     */
    @Bean("legacySeekWithDltHandler")
    @Deprecated
    public SeekToCurrentErrorHandler legacySeekWithDltHandler() {

        // DLT recoverer for failed messages
        DeadLetterPublishingRecoverer dltRecoverer = new
DeadLetterPublishingRecoverer(kafkaTemplate);

        // Exponential backoff
        ExponentialBackOff backOff = new ExponentialBackOff();
        backOff.setInitialInterval(1000L);
        backOff.setMultiplier(2.0);
        backOff.setMaxInterval(10000L);
        backOff.setMaxElapsedTime(60000L); // Max 1 minute

        SeekToCurrentErrorHandler errorHandler = new
SeekToCurrentErrorHandler(dltRecoverer, backOff);

        // Exception classification
        errorHandler.addNotRetryableExceptions(
            IllegalArgumentException.class,
            ClassCastException.class,

org.springframework.kafka.support.serializer.DeserializationException.class
        );

        errorHandler.addRetryableExceptions(
            ExternalServiceException.class,
            java.util.concurrent.TimeoutException.class
        );

        log.warn("Using deprecated SeekToCurrentErrorHandler with DLT - migrate to
DefaultErrorHandler");

        return errorHandler;
    }

    /**
     * Legacy container factory configuration (DEPRECATED)
     *
```

```
     * @deprecated Use setCommonErrorHandler instead of setErrorHandler
     */
    @Bean("legacyContainerFactory")
    @Deprecated
    public ConcurrentKafkaListenerContainerFactory<String, Object>
legacyContainerFactory() {

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

        factory.setConsumerFactory(consumerFactory());

        // OLD WAY - deprecated
        factory.setErrorHandler(legacySeekToCurrentErrorHandler());

        factory.setConcurrency(3);

        log.warn("Using deprecated setErrorHandler method - use
setCommonErrorHandler instead");

        return factory;
    }
}

/**
 * Migration utilities and patterns from SeekToCurrentErrorHandler to
DefaultErrorHandler
 */
@Component
@lombok.extern.slf4j.Slf4j
public class ErrorHandlerMigrationGuide {

    /**
     * Example: Migrating from SeekToCurrentErrorHandler to DefaultErrorHandler
     */
    public void migrationExample() {

        log.info("=== Migration Guide: SeekToCurrentErrorHandler ->
DefaultErrorHandler ===");

        // OLD WAY (Spring Kafka 2.7 and earlier)
        showLegacyConfiguration();

        // NEW WAY (Spring Kafka 2.8+)
        showModernConfiguration();

        log.info("=== Key Migration Points ===");
        showMigrationKeyPoints();
    }

    private void showLegacyConfiguration() {
        log.info("--- LEGACY Configuration (DEPRECATED) ---");

        String legacyCode = """
```

```
            // OLD WAY - Don't use this in new projects
            @Bean
            public ConcurrentKafkaListenerContainerFactory<String, Object>
containerFactory() {
                ConcurrentKafkaListenerContainerFactory<String, Object> factory =
                    new ConcurrentKafkaListenerContainerFactory<>();

                factory.setConsumerFactory(consumerFactory());

                // DEPRECATED: setErrorHandler
                factory.setErrorHandler(new SeekToCurrentErrorHandler(
                    new DeadLetterPublishingRecoverer(kafkaTemplate),
                    new FixedBackOff(1000L, 3L)
                ));

                return factory;
            }
            """;

        log.info("Legacy Configuration:\n{}", legacyCode);
    }

    private void showModernConfiguration() {
        log.info("--- MODERN Configuration (Recommended) ---");

        String modernCode = """
            // NEW WAY - Use this approach
            @Bean
            public ConcurrentKafkaListenerContainerFactory<String, Object>
containerFactory() {
                ConcurrentKafkaListenerContainerFactory<String, Object> factory =
                    new ConcurrentKafkaListenerContainerFactory<>();

                factory.setConsumerFactory(consumerFactory());

                // NEW: setCommonErrorHandler with DefaultErrorHandler
                factory.setCommonErrorHandler(new DefaultErrorHandler(
                    new DeadLetterPublishingRecoverer(kafkaTemplate),
                    new FixedBackOff(1000L, 3L)
                ));

                return factory;
            }
            """;

        log.info("Modern Configuration:\n{}", modernCode);
    }

    private void showMigrationKeyPoints() {
        String keyPoints = """
            1. Replace SeekToCurrentErrorHandler with DefaultErrorHandler
            2. Use setCommonErrorHandler() instead of setErrorHandler()
            3. DefaultErrorHandler works with both record and batch listeners
            4. Better performance with non-blocking retries (seekAfterError=false)
```

```
            5. Enhanced exception classification and retry listeners
            6. Improved observability and metrics

            Migration Checklist:
            ✓ Update Spring Kafka to 2.8+
            ✓ Replace SeekToCurrentErrorHandler with DefaultErrorHandler
            ✓ Change setErrorHandler to setCommonErrorHandler
            ✓ Test retry behavior (DefaultErrorHandler has different semantics)
            ✓ Update monitoring and metrics collection
            ✓ Review exception classification configuration
            """;

        log.info("Key Migration Points:\n{}", keyPoints);
    }

    /**
     * Automated migration helper for configuration
     */
    public DefaultErrorHandler migrateFromSeekToCurrent(SeekToCurrentErrorHandler
legacyHandler) {
        log.info("Performing automated migration from SeekToCurrentErrorHandler");

        try {
            // Extract configuration from legacy handler using reflection
            BackOff backOff = extractBackOffFromLegacy(legacyHandler);
            ConsumerRecordRecoverer recoverer =
extractRecovererFromLegacy(legacyHandler);

            // Create new DefaultErrorHandler
            DefaultErrorHandler newHandler = new DefaultErrorHandler(recoverer,
backOff);

            // Migrate exception classifications
            migrateExceptionClassifications(legacyHandler, newHandler);

            log.info("Migration completed successfully");

            return newHandler;

        } catch (Exception e) {
            log.error("Migration failed, manual configuration required", e);
            throw new RuntimeException("Migration failed", e);
        }
    }

    private BackOff extractBackOffFromLegacy(SeekToCurrentErrorHandler
legacyHandler) {
        // In real implementation, would use reflection to extract backoff
        // For demo, return default
        return new FixedBackOff(1000L, 3L);
    }

    private ConsumerRecordRecoverer
extractRecovererFromLegacy(SeekToCurrentErrorHandler legacyHandler) {
```

```
        // In real implementation, would extract recoverer
        // For demo, return logging recoverer
        return (record, exception) -> {
            log.error("Migrated recoverer: topic={}, offset={}, error={}",
                record.topic(), record.offset(), exception.getMessage());
        };
    }

    private void migrateExceptionClassifications(SeekToCurrentErrorHandler legacy,
                                                 DefaultErrorHandler modern) {
        // Copy exception classifications from legacy to modern handler
        // This would require reflection to access private fields in real
implementation

        // For demo, set common classifications
        modern.addNotRetryableExceptions(
            IllegalArgumentException.class,

org.springframework.kafka.support.serializer.DeserializationException.class
        );

        modern.addRetryableExceptions(
            ExternalServiceException.class,
            java.util.concurrent.TimeoutException.class
        );

        log.info("Migrated exception classifications");
    }
}

/**
 * Side-by-side comparison service for testing migration
 */
@Service
@lombok.extern.slf4j.Slf4j
public class MigrationTestingService {

    @Autowired
    private KafkaTemplate<String, Object> kafkaTemplate;

    /**
     * Test both legacy and modern error handlers with same scenario
     */
    public void compareBehavior() {
        log.info("Starting behavior comparison between legacy and modern error
handlers");

        // Test scenarios
        List<TestScenario> scenarios = createTestScenarios();

        for (TestScenario scenario : scenarios) {
            log.info("Testing scenario: {}", scenario.getName());

            // Test with legacy handler (for comparison only)
```

```
            testWithLegacyHandler(scenario);

            // Test with modern handler
            testWithModernHandler(scenario);

            compareResults(scenario);
        }
    }

    private List<TestScenario> createTestScenarios() {
        return Arrays.asList(
            TestScenario.builder()
                .name("Validation Exception")
                .exception(new ValidationException("Invalid data"))
                .expectedRetries(0) // Non-retryable
                .build(),

            TestScenario.builder()
                .name("External Service Exception")
                .exception(new ExternalServiceException("Service unavailable"))
                .expectedRetries(3) // Retryable
                .build(),

            TestScenario.builder()
                .name("Deserialization Exception")
                .exception(new
org.springframework.kafka.support.serializer.DeserializationException(
                    "Cannot deserialize", null, false, null))
                .expectedRetries(0) // Non-retryable
                .build()
        );
    }

    private void testWithLegacyHandler(TestScenario scenario) {
        log.debug("Testing legacy handler with scenario: {}", scenario.getName());
        // Implementation would test SeekToCurrentErrorHandler behavior
    }

    private void testWithModernHandler(TestScenario scenario) {
        log.debug("Testing modern handler with scenario: {}", scenario.getName());
        // Implementation would test DefaultErrorHandler behavior
    }

    private void compareResults(TestScenario scenario) {
        log.info("Scenario '{}' comparison completed", scenario.getName());
        // Implementation would compare and report differences
    }
}

// Supporting classes for migration testing
@lombok.Data
@lombok.Builder
@lombok.NoArgsConstructor
@lombok.AllArgsConstructor
```

```
class TestScenario {
    private String name;
    private Exception exception;
    private int expectedRetries;
    private boolean expectDlt;
}
```

📊 Comparisons & Trade-offs

Error Handler Comparison Matrix

Aspect	SeekToCurrentErrorHandler	DefaultErrorHandler	Custom Handler
Status	❌ Deprecated	☑ Current	☑ Advanced
Retry Mechanism	Seek to offset	Seek or non-blocking	Configurable
Performance	★★★	★★★★★	★★★★
Flexibility	★★	★★★★	★★★★★
Complexity	★★	★★★	★★★★★
Batch Support	❌ Limited	☑ Full	☑ Custom
Non-blocking Retries	❌ No	☑ Yes	☑ Configurable
Observability	★★	★★★★	★★★★★

Backoff Policy Performance Comparison

Policy Type	Memory Usage	CPU Impact	Best Use Case	Thundering Herd Risk
Fixed	★★★★★	★★★★★	Predictable failures	★★★
Exponential	★★★★	★★★★	Transient failures	★★
Jittered	★★★	★★★	High concurrency	★
Adaptive	★★	★★	Dynamic load	★

Recovery Strategy Trade-offs

Strategy	Data Loss Risk	Processing Delay	Operational Overhead	Use Case
Log Only	🟤 High	🟢 None	🟢 Low	Development/Testing
DLT	🟢 None	🟡 Medium	🟡 Medium	Production
Retry Topic	🟢 None	🟤 High	🟤 High	Critical Messages
Manual Review	🟢 None	🟤 Very High	🟤 Very High	Complex Failures

Common Pitfalls & Best Practices

Critical Anti-Patterns to Avoid

❌ Configuration Mistakes

```
// DON'T - Infinite retries without circuit breaker
@Bean
public CommonErrorHandler badInfiniteRetryHandler() {
    // BAD: This will retry forever and can bring down the system
    ExponentialBackOff infiniteBackOff = new ExponentialBackOff();
    infiniteBackOff.setMaxElapsedTime(Long.MAX_VALUE); // NEVER DO THIS

    return new DefaultErrorHandler(infiniteBackOff);
}

// DON'T - Ignoring poison pills
@Bean
public CommonErrorHandler badPoisonPillHandler() {
    DefaultErrorHandler handler = new DefaultErrorHandler();

    // BAD: Not classifying deserialization exceptions as non-retryable
    // This will cause infinite retries on poison pills
    handler.addRetryableExceptions(

org.springframework.kafka.support.serializer.DeserializationException.class
    );

    return handler;
}

// DON'T - Blocking operations in error handlers
public class BadErrorHandler implements ConsumerRecordRecoverer {

    @Override
    public void accept(ConsumerRecord<?, ?> record, Exception exception) {
        try {
            // BAD: Synchronous HTTP call in error handler
            String response = restTemplate.postForObject("http://error-service",
                record.value(), String.class);

            // BAD: Blocking database write
            errorRepository.save(new ErrorRecord(record, exception));

        } catch (Exception e) {
            // BAD: Nested exception handling without bounds
            throw new RuntimeException(e);
        }
    }
}
```

❌ Resource Management Issues

```
// DON'T - Memory leaks in custom handlers
public class LeakyErrorHandler implements ConsumerRecordRecoverer {

    // BAD: Growing without bounds
    private final Map<String, List<Exception>> errorHistory = new HashMap<>();

    @Override
    public void accept(ConsumerRecord<?, ?> record, Exception exception) {
        String key = record.topic() + "-" + record.partition();

        // BAD: Never cleaned up - will cause OutOfMemoryError
        errorHistory.computeIfAbsent(key, k -> new ArrayList<>()).add(exception);

        log.error("Error count for {}: {}", key, errorHistory.get(key).size());
    }
}

// DON'T - Thread pool mismanagement
@Component
public class BadAsyncErrorHandler {

    // BAD: Unbounded thread pool
    private final ExecutorService executor = Executors.newCachedThreadPool();

    public void handleErrorAsync(ConsumerRecord<?, ?> record, Exception exception)
{
        // BAD: No bounded queue, no proper shutdown
        executor.submit(() -> {
            // Long running task without timeout
            processErrorSlowly(record, exception);
        });
    }

    // Missing @PreDestroy to shutdown executor properly
}
```

Production Best Practices

☑ Optimal Error Handler Configuration

```
/**
 * ☑ GOOD - Production-ready error handler configuration
 */
@Configuration
@lombok.extern.slf4j.Slf4j
public class ProductionErrorHandlerConfiguration {

    @Autowired
```

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

    @Autowired
    private MeterRegistry meterRegistry;

    @Bean
    public CommonErrorHandler productionErrorHandler() {

        // Smart DLT routing based on failure analysis
        DeadLetterPublishingRecoverer recoverer = new
DeadLetterPublishingRecoverer(
            kafkaTemplate, this::intelligentDltRouting);

        // Enhanced headers for debugging and recovery
        recoverer.setHeadersFunction(this::createComprehensiveHeaders);

        // Exponential backoff with reasonable limits
        ExponentialBackOffWithMaxRetries backOff = new
ExponentialBackOffWithMaxRetries(6);
        backOff.setInitialInterval(1000L);      // Start with 1 second
        backOff.setMultiplier(1.5);             // Gentle escalation
        backOff.setMaxInterval(30000L);         // Cap at 30 seconds

        DefaultErrorHandler errorHandler = new DefaultErrorHandler(recoverer,
backOff);

        // Comprehensive exception classification
        configureProductionExceptions(errorHandler);

        // Advanced retry listeners with circuit breaker
        errorHandler.setRetryListeners(createProductionRetryListener());

        // Reset state on exception change for better recovery
        errorHandler.setResetStateOnExceptionChange(true);

        log.info("Configured production error handler: maxRetries=6, backoff=
[1s,1.5s,2.25s,3.38s,5.06s,7.59s]");

        return errorHandler;
    }

    private TopicPartition intelligentDltRouting(ConsumerRecord<?, ?> record,
Exception exception) {
        String originalTopic = record.topic();
        int originalPartition = record.partition();

        // Route based on failure analysis
        FailureCategory category = analyzeFailure(exception, record);

        return switch (category) {
            case POISON_PILL -> new TopicPartition(originalTopic + ".poison.DLT",
originalPartition);
            case BUSINESS_RULE -> new TopicPartition(originalTopic +
".business.DLT", originalPartition);
```

```
            case EXTERNAL_SERVICE -> new TopicPartition(originalTopic +
".external.DLT", originalPartition);
            case INFRASTRUCTURE -> new TopicPartition(originalTopic +
".infra.DLT", originalPartition);
            default -> new TopicPartition(originalTopic + ".DLT",
originalPartition);
        };
    }

    private Map<String, Object> createComprehensiveHeaders(ConsumerRecord<?, ?>
record, Exception exception) {
        Map<String, Object> headers = new HashMap<>();

        // Standard DLT headers
        headers.put(KafkaHeaders.DLT_ORIGINAL_TOPIC, record.topic());
        headers.put(KafkaHeaders.DLT_ORIGINAL_PARTITION, record.partition());
        headers.put(KafkaHeaders.DLT_ORIGINAL_OFFSET, record.offset());
        headers.put(KafkaHeaders.DLT_ORIGINAL_TIMESTAMP, record.timestamp());
        headers.put(KafkaHeaders.DLT_EXCEPTION_FQCN,
exception.getClass().getName());
        headers.put(KafkaHeaders.DLT_EXCEPTION_MESSAGE,
            truncateString(exception.getMessage(), 1000));

        // Enhanced diagnostic information
        headers.put("failure-timestamp", System.currentTimeMillis());
        headers.put("failure-host", getHostname());
        headers.put("failure-application", getApplicationInfo());
        headers.put("failure-category", analyzeFailure(exception, record).name());
        headers.put("recovery-priority", determinePriority(record.value()));
        headers.put("business-context", extractBusinessContext(record.value()));

        // Processing metrics
        headers.put("retry-count", getRetryCount(record));
        headers.put("processing-duration-ms", getProcessingDuration(record));
        headers.put("message-age-ms", System.currentTimeMillis() -
record.timestamp());

        // Exception analysis
        headers.put("root-cause", getRootCause(exception).getClass().getName());
        headers.put("exception-chain-depth", getExceptionChainDepth(exception));

        // Stack trace (compressed)
        String stackTrace = getCompressedStackTrace(exception);
        headers.put(KafkaHeaders.DLT_EXCEPTION_STACKTRACE, stackTrace);

        return headers;
    }

    private void configureProductionExceptions(DefaultErrorHandler errorHandler) {
        // Fatal exceptions - never retry
        errorHandler.addNotRetryableExceptions(
            // Serialization/Deserialization issues

org.springframework.kafka.support.serializer.DeserializationException.class,
```

```
org.springframework.messaging.converter.MessageConversionException.class,
            org.springframework.core.convert.ConversionException.class,

            // Programming errors
            IllegalArgumentException.class,
            NullPointerException.class,
            ClassCastException.class,
            NoSuchMethodException.class,

            // Security issues
            SecurityException.class,
            java.nio.file.AccessDeniedException.class,

            // Resource issues that won't resolve with retry
            OutOfMemoryError.class,
            StackOverflowError.class
        );

        // Retryable exceptions - might resolve
        errorHandler.addRetryableExceptions(
            // Network/External service issues
            ExternalServiceException.class,
            java.util.concurrent.TimeoutException.class,
            java.net.ConnectException.class,
            java.net.SocketTimeoutException.class,

            // Database issues
            org.springframework.dao.TransientDataAccessException.class,
            org.springframework.dao.QueryTimeoutException.class,

            // Business logic that might resolve
            TemporaryBusinessException.class,
            RateLimitExceededException.class
        );
    }

    private RetryListener createProductionRetryListener() {
        return new ProductionRetryListener(meterRegistry);
    }

    // Analysis and utility methods
    private FailureCategory analyzeFailure(Exception exception, ConsumerRecord<?,
?> record) {
        if (exception instanceof
org.springframework.kafka.support.serializer.DeserializationException ||
            exception instanceof ClassCastException) {
            return FailureCategory.POISON_PILL;
        }

        if (exception instanceof ValidationException ||
            exception instanceof BusinessRuleException) {
            return FailureCategory.BUSINESS_RULE;
        }
```

```
        if (exception instanceof ExternalServiceException ||
            exception instanceof java.util.concurrent.TimeoutException) {
            return FailureCategory.EXTERNAL_SERVICE;
        }

        if (exception instanceof org.springframework.dao.DataAccessException) {
            return FailureCategory.INFRASTRUCTURE;
        }

        return FailureCategory.UNKNOWN;
    }

    private String determinePriority(Object messageValue) {
        if (messageValue instanceof OrderEvent order) {
            // High priority for large orders
            if (order.getAmount().compareTo(new java.math.BigDecimal("1000")) > 0)
{
                return "HIGH";
            }
        } else if (messageValue instanceof PaymentEvent) {
            return "HIGH"; // All payments are high priority
        }

        return "MEDIUM";
    }

    private String extractBusinessContext(Object messageValue) {
        if (messageValue instanceof OrderEvent order) {
            return String.format("order:%s,customer:%s", order.getOrderId(),
order.getCustomerId());
        } else if (messageValue instanceof PaymentEvent payment) {
            return String.format("payment:%s,method:%s", payment.getPaymentId(),
payment.getMethod());
        }

        return "unknown";
    }

    private String getCompressedStackTrace(Exception exception) {
        String fullTrace = getStackTraceString(exception);

        // Compress stack trace by removing common framework lines
        String[] lines = fullTrace.split("\n");
        StringBuilder compressed = new StringBuilder();

        for (String line : lines) {
            // Keep application lines and key framework lines
            if (line.contains("com.example") || // Application packages
                line.contains("Caused by:") ||
                line.contains("org.springframework.kafka") ||
                compressed.length() < 2000) { // Keep first 2000 chars regardless

                compressed.append(line).append("\n");
```

```
            }
        }

        return compressed.toString();
    }

    // Utility methods
    private String truncateString(String str, int maxLength) {
        if (str == null) return null;
        return str.length() > maxLength ? str.substring(0, maxLength) + "..." :
str;
    }

    private String getHostname() {
        try {
            return InetAddress.getLocalHost().getHostName();
        } catch (Exception e) {
            return "unknown";
        }
    }

    private String getApplicationInfo() {
        return "kafka-processor-v1.0"; // Could read from properties
    }

    private int getRetryCount(ConsumerRecord<?, ?> record) {
        Header retryHeader = record.headers().lastHeader("retry-count");
        return retryHeader != null ? ByteBuffer.wrap(retryHeader.value()).getInt()
: 0;
    }

    private long getProcessingDuration(ConsumerRecord<?, ?> record) {
        Header startHeader = record.headers().lastHeader("processing-start");
        if (startHeader != null) {
            long startTime = ByteBuffer.wrap(startHeader.value()).getLong();
            return System.currentTimeMillis() - startTime;
        }
        return 0;
    }

    private Throwable getRootCause(Throwable throwable) {
        Throwable cause = throwable;
        while (cause.getCause() != null) {
            cause = cause.getCause();
        }
        return cause;
    }

    private int getExceptionChainDepth(Throwable throwable) {
        int depth = 0;
        Throwable cause = throwable;
        while (cause != null) {
            depth++;
            cause = cause.getCause();
```

```
        }
        return depth;
    }

    private String getStackTraceString(Exception exception) {
        StringWriter sw = new StringWriter();
        PrintWriter pw = new PrintWriter(sw);
        exception.printStackTrace(pw);
        return sw.toString();
    }
}

/**
 * ☑ GOOD - Production retry listener with circuit breaker
 */
@Component
public class ProductionRetryListener implements RetryListener {

    private final MeterRegistry meterRegistry;
    private final Map<String, CircuitBreaker> circuitBreakers = new
ConcurrentHashMap<>();

    public ProductionRetryListener(MeterRegistry meterRegistry) {
        this.meterRegistry = meterRegistry;
    }

    @Override
    public void failedDelivery(ConsumerRecord<?, ?> record, Exception ex, int
deliveryAttempt) {
        String topic = record.topic();
        String exceptionType = ex.getClass().getSimpleName();

        // Update retry metrics
        meterRegistry.counter("kafka.error.retry.attempts",
            Tags.of(
                "topic", topic,
                "exception", exceptionType,
                "attempt", String.valueOf(deliveryAttempt)
            )).increment();

        // Update circuit breaker
        getCircuitBreaker(topic).recordFailure();

        // Log with appropriate level based on attempt count
        if (deliveryAttempt <= 2) {
            log.debug("Retry attempt {} for topic {}: {}", deliveryAttempt, topic,
ex.getMessage());
        } else if (deliveryAttempt <= 4) {
            log.warn("Retry attempt {} for topic {}: {}", deliveryAttempt, topic,
ex.getMessage());
        } else {
            log.error("High retry attempt {} for topic {}: {}", deliveryAttempt,
topic, ex.getMessage());
```

```
            // Alert on high retry counts
            if (deliveryAttempt >= 5) {
                alertHighRetryCount(record, ex, deliveryAttempt);
            }
        }

        // Check circuit breaker state
        if (getCircuitBreaker(topic).getState() == CircuitBreaker.State.OPEN) {
            log.error("Circuit breaker OPEN for topic: {}", topic);
            alertCircuitBreakerOpen(topic);
        }
    }

    @Override
    public void recovered(ConsumerRecord<?, ?> record, Exception ex) {
        String topic = record.topic();
        String exceptionType = ex.getClass().getSimpleName();

        // Update recovery metrics
        meterRegistry.counter("kafka.error.recovered",
            Tags.of("topic", topic, "exception", exceptionType))
            .increment();

        // Update circuit breaker
        getCircuitBreaker(topic).recordSuccess();

        log.info("Message recovered after retries: topic={}, partition={}, offset=
{}",
            topic, record.partition(), record.offset());
    }

    @Override
    public void recoveryFailed(ConsumerRecord<?, ?> record, Exception original,
Exception failure) {
        String topic = record.topic();
        String originalException = original.getClass().getSimpleName();

        // Update failure metrics
        meterRegistry.counter("kafka.error.recovery.failed",
            Tags.of("topic", topic, "exception", originalException))
            .increment();

        log.error("Message recovery failed, sending to DLT: topic={}, partition=
{}, offset={}, original={}, recovery={}",
            topic, record.partition(), record.offset(),
            original.getMessage(), failure.getMessage());

        // Alert on recovery failures for critical topics
        if (isCriticalTopic(topic)) {
            alertRecoveryFailure(record, original, failure);
        }
    }

    private CircuitBreaker getCircuitBreaker(String topic) {
```

```
        return circuitBreakers.computeIfAbsent(topic, t -> {
            CircuitBreakerConfig config = CircuitBreakerConfig.custom()
                .failureRateThreshold(50) // 50% failure rate
                .waitDurationInOpenState(Duration.ofMinutes(2))
                .slidingWindowSize(20)
                .minimumNumberOfCalls(10)
                .build();

            return CircuitBreaker.of(topic + "-circuit-breaker", config);
        });
    }

    private void alertHighRetryCount(ConsumerRecord<?, ?> record, Exception ex,
int attempt) {
        log.error("🚨 HIGH RETRY COUNT ALERT: topic={}, attempt={}, error={}",
            record.topic(), attempt, ex.getMessage());
        // Integration with alerting systems
    }

    private void alertCircuitBreakerOpen(String topic) {
        log.error("🚨 CIRCUIT BREAKER OPEN: topic={}", topic);
        // Integration with alerting systems
    }

    private void alertRecoveryFailure(ConsumerRecord<?, ?> record, Exception
original, Exception failure) {
        log.error("🚨 RECOVERY FAILURE ALERT: topic={}, originalError={},
recoveryError={}",
            record.topic(), original.getMessage(), failure.getMessage());
        // Integration with alerting systems
    }

    private boolean isCriticalTopic(String topic) {
        return topic.contains("payment") || topic.contains("order") ||
topic.contains("critical");
    }
}

// Supporting enums and classes
enum FailureCategory {
    POISON_PILL,
    BUSINESS_RULE,
    EXTERNAL_SERVICE,
    INFRASTRUCTURE,
    UNKNOWN
}

class TemporaryBusinessException extends Exception {
    public TemporaryBusinessException(String message) { super(message); }
}

class RateLimitExceededException extends Exception {
    public RateLimitExceededException(String message) { super(message); }
}
```

☑ Health Monitoring and Observability

```
/**
 * ☑ GOOD - Comprehensive error handling monitoring
 */
@Component
@lombok.extern.slf4j.Slf4j
public class ErrorHandlingMonitoring {

    @Autowired
    private MeterRegistry meterRegistry;

    @Autowired
    private KafkaListenerEndpointRegistry endpointRegistry;

    /**
     * Monitor error handler health and performance
     */
    @Scheduled(fixedDelay = 60000) // Every minute
    public void monitorErrorHandlingHealth() {
        Collection<MessageListenerContainer> containers =
endpointRegistry.getAllListenerContainers();

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

            // Check container health
            boolean isRunning = container.isRunning();
            boolean isPaused = container.isContainerPaused();

            // Update health metrics
            meterRegistry.gauge("kafka.container.health",
                Tags.of("listener.id", listenerId, "state",
                    isRunning ? (isPaused ? "paused" : "running") : "stopped"),
                isRunning && !isPaused ? 1.0 : 0.0);

            // Alert on unhealthy containers
            if (!isRunning) {
                log.error("🚨 Container not running: {}", listenerId);
                alertUnhealthyContainer(listenerId, "stopped");
            } else if (isPaused) {
                log.warn("⚠ Container paused: {}", listenerId);
                alertUnhealthyContainer(listenerId, "paused");
            }
        }
    }

    /**
     * Monitor DLT topic growth rates
     */
```

```
    @Scheduled(fixedDelay = 300000) // Every 5 minutes
    public void monitorDltGrowth() {
        List<String> dltTopics = getDltTopics();

        for (String dltTopic : dltTopics) {
            try {
                long messageCount = getDltMessageCount(dltTopic);

                // Track DLT message count
                meterRegistry.gauge("kafka.dlt.message.count",
                    Tags.of("topic", dltTopic), messageCount);

                // Alert on rapid DLT growth
                if (isRapidGrowth(dltTopic, messageCount)) {
                    alertRapidDltGrowth(dltTopic, messageCount);
                }

            } catch (Exception e) {
                log.error("Failed to monitor DLT topic: {}", dltTopic, e);
            }
        }
    }

    /**
     * Generate error handling reports
     */
    @Scheduled(cron = "0 0 * * * *") // Every hour
    public void generateErrorReport() {
        try {
            ErrorHandlingReport report = createErrorReport();

            log.info("Error Handling Report: {}", report);

            // Send report to monitoring systems
            sendReportToMonitoring(report);

            // Alert on concerning trends
            if (report.hasAnomalies()) {
                alertErrorAnomalies(report);
            }

        } catch (Exception e) {
            log.error("Failed to generate error report", e);
        }
    }

    private void alertUnhealthyContainer(String listenerId, String state) {
        log.error("Container health alert: listener={}, state={}", listenerId,
state);
    }

    private List<String> getDltTopics() {
        // Implementation would query Kafka admin client
        return Arrays.asList("orders.DLT", "payments.DLT", "notifications.DLT");
```

```
    }

    private long getDltMessageCount(String dltTopic) {
        // Implementation would query Kafka admin client for topic message count
        return 0; // Placeholder
    }

    private boolean isRapidGrowth(String dltTopic, long currentCount) {
        // Implementation would compare with historical data
        return false; // Placeholder
    }

    private void alertRapidDltGrowth(String dltTopic, long messageCount) {
        log.error("🚨 RAPID DLT GROWTH: topic={}, count={}", dltTopic,
messageCount);
    }

    private ErrorHandlingReport createErrorReport() {
        // Implementation would aggregate metrics and create report
        return new ErrorHandlingReport();
    }

    private void sendReportToMonitoring(ErrorHandlingReport report) {
        // Send to monitoring systems
    }

    private void alertErrorAnomalies(ErrorHandlingReport report) {
        log.error("🚨 ERROR HANDLING ANOMALIES DETECTED: {}",
report.getAnomalies());
    }

    static class ErrorHandlingReport {
        public boolean hasAnomalies() { return false; }
        public List<String> getAnomalies() { return List.of(); }
    }
}
```

🌍 Real-World Use Cases

E-commerce Platform Error Handling

```
/**
 * Production e-commerce error handling patterns
 */
@Service
@lombok.extern.slf4j.Slf4j
public class EcommerceErrorHandlingService {

    /**
     * Order processing with tiered error handling
```

```
     */
    @KafkaListener(
        topics = "orders",
        groupId = "order-processor",
        containerFactory = "resilientOrderContainerFactory"
    )
    public void processOrder(@Payload OrderEvent order,
                             Acknowledgment ack) {

        String orderId = order.getOrderId();
        log.info("Processing order: orderId={}, customerId={}, amount={}",
            orderId, order.getCustomerId(), order.getAmount());

        try {
            // Critical order processing
            validateOrder(order);
            checkInventory(order);
            processPayment(order);
            createShipment(order);

            ack.acknowledge();
            log.info("Order processed successfully: {}", orderId);

        } catch (ValidationException e) {
            log.error("Order validation failed: orderId={}, error={}", orderId,
e.getMessage());
            // Will go to orders.business.DLT for manual review
            throw e;

        } catch (InventoryException e) {
            log.warn("Inventory issue: orderId={}, error={}", orderId,
e.getMessage());
            // Will be retried - inventory might become available
            throw new TemporaryBusinessException("Inventory temporarily
unavailable", e);

        } catch (PaymentException e) {
            log.error("Payment failed: orderId={}, error={}", orderId,
e.getMessage());
            // Critical - goes to high-priority DLT
            throw e;

        } catch (ExternalServiceException e) {
            log.warn("External service error: orderId={}, service={}, error={}",
                orderId, e.getServiceName(), e.getMessage());
            // Will be retried with exponential backoff
            throw e;

        } catch (Exception e) {
            log.error("Unexpected error processing order: orderId={}", orderId,
e);
            throw e;
        }
    }
```

```
    /**
     * Payment processing with immediate retry for specific errors
     */
    @KafkaListener(
        topics = "payments",
        groupId = "payment-processor",
        containerFactory = "highPriorityContainerFactory"
    )
    public void processPayment(@Payload PaymentEvent payment,
                               Acknowledgment ack) {

        String paymentId = payment.getPaymentId();
        log.info("Processing payment: paymentId={}, amount={}, method={}",
            paymentId, payment.getAmount(), payment.getMethod());

        try {
            // Payment processing is critical
            validatePayment(payment);

            // Call payment gateway
            PaymentResult result = paymentGateway.processPayment(payment);

            if (result.isSuccessful()) {
                updateOrderStatus(payment.getOrderId(), "PAID");
                ack.acknowledge();
                log.info("Payment processed successfully: {}", paymentId);
            } else {
                throw new PaymentException("Payment declined: " +
result.getDeclineReason());
            }

        } catch (PaymentValidationException e) {
            log.error("Payment validation failed: paymentId={}, error={}",
paymentId, e.getMessage());
            // Invalid payment data - goes to manual review
            throw e;

        } catch (PaymentGatewayException e) {
            if (e.isRetryable()) {
                log.warn("Retryable payment gateway error: paymentId={}, error=
{}",
                    paymentId, e.getMessage());
                throw new ExternalServiceException("Payment gateway", e);
            } else {
                log.error("Non-retryable payment gateway error: paymentId={},
error={}",
                    paymentId, e.getMessage());
                throw new PaymentException("Payment gateway rejected", e);
            }

        } catch (Exception e) {
            log.error("Unexpected payment processing error: paymentId={}",
paymentId, e);
```

```
                throw e;
            }
        }

        // Business logic methods
        private void validateOrder(OrderEvent order) throws ValidationException {
            if (order.getOrderId() == null || order.getOrderId().isEmpty()) {
                throw new ValidationException("Order ID is required");
            }
            if (order.getAmount().compareTo(java.math.BigDecimal.ZERO) <= 0) {
                throw new ValidationException("Order amount must be positive");
            }
        }

        private void checkInventory(OrderEvent order) throws InventoryException {
            // Simulate inventory check that might fail temporarily
            if (Math.random() < 0.1) { // 10% chance of temporary inventory issue
                throw new InventoryException("Temporary inventory system
unavailable");
            }
        }

        private void processPayment(OrderEvent order) throws PaymentException {
            // Simulate payment processing
            if (order.getAmount().compareTo(new java.math.BigDecimal("10000")) > 0) {
                throw new PaymentException("Amount exceeds daily limit");
            }
        }

        private void createShipment(OrderEvent order) throws ExternalServiceException
{
            // Simulate shipping service call
            if (Math.random() < 0.05) { // 5% chance of shipping service error
                throw new ExternalServiceException("shipping-service", "Service
temporarily unavailable");
            }
        }

        private void validatePayment(PaymentEvent payment) throws
PaymentValidationException {
            if (payment.getAmount().compareTo(java.math.BigDecimal.ZERO) <= 0) {
                throw new PaymentValidationException("Payment amount must be
positive");
            }
        }

        private void updateOrderStatus(String orderId, String status) {
            log.debug("Updating order status: orderId={}, status={}", orderId,
status);
        }

        // Mock services
        @Autowired private PaymentGateway paymentGateway;
}
```

Financial Services with Exactly-Once Processing

```
/**
 * Financial transaction processing with strict error handling
 */
@Service
@Transactional
public class FinancialTransactionProcessor {

    /**
     * Money transfer processing with idempotency and exact error handling
     */
    @KafkaListener(
        topics = "money-transfers",
        groupId = "transfer-processor",
        containerFactory = "exactlyOnceContainerFactory"
    )
    public void processMoneyTransfer(@Payload MoneyTransferEvent transfer,
                                     Acknowledgment ack) {

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

        try {
            // Idempotency check
            if (isAlreadyProcessed(transferId)) {
                log.info("Transfer already processed: {}", transferId);
                ack.acknowledge();
                return;
            }

            // Strict validation for financial data
            validateTransfer(transfer);

            // Check account balances
            if (!hasSufficientBalance(transfer.getFromAccount(),
transfer.getAmount())) {
                throw new InsufficientFundsException("Insufficient balance in
account: " + transfer.getFromAccount());
            }

            // Execute atomic transfer
            executeTransfer(transfer);

            // Mark as processed
            markAsProcessed(transferId);
```

```
            ack.acknowledge();
            log.info("Money transfer completed successfully: {}", transferId);

        } catch (ValidationException | InsufficientFundsException e) {
            log.error("Transfer validation/balance error (non-retryable):
transferId={}, error={}",
                transferId, e.getMessage());
            // These go to manual review - financial data issues
            throw e;

        } catch (DatabaseException e) {
            log.warn("Database error during transfer (retryable): transferId={},
error={}",
                transferId, e.getMessage());
            // Database issues can be retried
            throw e;

        } catch (Exception e) {
            log.error("Unexpected error during transfer: transferId={}",
transferId, e);
            // Unknown errors in financial processing are critical
            throw new CriticalFinancialException("Critical error in transfer
processing", e);
        }
    }

    private boolean isAlreadyProcessed(String transferId) {
        // Check idempotency store
        return false; // Placeholder
    }

    private void validateTransfer(MoneyTransferEvent transfer) throws
ValidationException {
        if (transfer.getAmount().compareTo(java.math.BigDecimal.ZERO) <= 0) {
            throw new ValidationException("Transfer amount must be positive");
        }
        if (transfer.getFromAccount().equals(transfer.getToAccount())) {
            throw new ValidationException("Cannot transfer to same account");
        }
    }

    private boolean hasSufficientBalance(String account, java.math.BigDecimal
amount) {
        // Check account balance
        return true; // Placeholder
    }

    private void executeTransfer(MoneyTransferEvent transfer) throws
DatabaseException {
        // Execute atomic database transaction
        log.debug("Executing transfer: {}", transfer.getTransferId());
    }

    private void markAsProcessed(String transferId) {
```

```
            // Mark in idempotency store
            log.debug("Marking transfer as processed: {}", transferId);
        }
    }
```

Version Highlights

Spring Kafka Error Handling Evolution

Version	Release	Key Error Handling Features
3.1.x	2024	Enhanced error handler observability, improved DLT routing
3.0.x	2023	Performance improvements, native compilation support
2.9.x	2022	Non-blocking retries, enhanced retry logic
2.8.x	2022	**DefaultErrorHandler introduction**, CommonErrorHandler interface
2.7.x	2021	Enhanced DLT headers, better batch error handling
2.6.x	2021	RetryableTopic annotation, non-blocking retries
2.5.x	2020	@DltHandler annotation, improved error recovery
2.4.x	2020	SeekToCurrentErrorHandler enhancements
2.3.x	2019	DeadLetterPublishingRecoverer improvements
2.2.x	2018	SeekToCurrentErrorHandler introduction

Modern Error Handling Features (2022-2025)

Spring Kafka 2.8+ Error Handling Revolution:

- **DefaultErrorHandler**: Unified error handler for record and batch listeners
- **CommonErrorHandler Interface**: Single interface for all error handling
- **Non-blocking Retries**: Better performance with `seekAfterError=false`
- **Enhanced Exception Classification**: More sophisticated retry/no-retry logic
- **Improved Observability**: Better metrics and monitoring capabilities

Key Migration Path:

```
Legacy (≤2.7)              Modern (2.8+)
─────────────────          ──────────────
SeekToCurrentErrorHandler → DefaultErrorHandler
setErrorHandler()         → setCommonErrorHandler()
Limited batch support     → Full batch support
Blocking retries          → Non-blocking retries
Basic metrics             → Enhanced observability
```

CLI Commands and Monitoring

Kafka CLI for Error Handling Monitoring

```
# Monitor DLT topics
kafka-topics --bootstrap-server localhost:9092 --list | grep -E "\.DLT$"

# Check DLT message counts
kafka-run-class kafka.tools.GetOffsetShell \
  --broker-list localhost:9092 \
  --topic orders.DLT \
  --time -1

# Consume from DLT for inspection
kafka-console-consumer \
  --bootstrap-server localhost:9092 \
  --topic orders.DLT \
  --from-beginning \
  --property print.headers=true \
  --property print.key=true

# Monitor consumer group lag (important for error handling)
kafka-consumer-groups \
  --bootstrap-server localhost:9092 \
  --group order-processor \
  --describe

# Check consumer group status during error scenarios
kafka-consumer-groups \
  --bootstrap-server localhost:9092 \
  --group order-processor \
  --describe \
  --verbose

# Reset consumer group offset (for retry scenarios)
kafka-consumer-groups \
  --bootstrap-server localhost:9092 \
  --group order-processor \
  --reset-offsets \
  --to-earliest \
  --topic orders \
  --execute

# Create DLT topics with proper configuration
kafka-topics \
  --bootstrap-server localhost:9092 \
  --create \
  --topic orders.DLT \
  --partitions 3 \
  --replication-factor 3 \
  --config cleanup.policy=compact \
  --config retention.ms=2592000000  # 30 days
```

JMX Metrics for Error Handling Monitoring

```
# Key JMX metrics to monitor for error handling
kafka.consumer:type=consumer-fetch-manager-metrics,client-id=*
kafka.consumer:type=consumer-coordinator-metrics,client-id=*

# Spring Kafka specific metrics
org.springframework.kafka:type=KafkaListenerContainerFactory,name=*
org.springframework.kafka:type=MessageListenerContainer,name=*

# Application-specific error metrics (with Micrometer)
application.kafka.error.retry.attempts
application.kafka.error.recovered
application.kafka.error.failed
application.kafka.dlt.published
application.kafka.container.health
```

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**: Always start with DefaultErrorHandler for new projects, configure appropriate exception classifications, use Dead Letter Topics for message preservation, implement comprehensive monitoring and alerting, and plan for manual recovery processes. The evolution from SeekToCurrentErrorHandler to DefaultErrorHandler represents a significant improvement in both performance and functionality - migrate legacy implementations when possible.

This completes the comprehensive Spring Kafka Error Handling & Retry guide with production-ready patterns, best practices, and extensive Java examples for building resilient Kafka-based applications.

[578] [579] [580]