Spring Kafka Error Handling & Retry: Part 2 - Dead Letter Topics & Legacy Handlers

Continuation of the comprehensive guide covering Dead Letter Topics configuration, retry patterns, and legacy SeekToCurrentErrorHandler migration strategies.

💀 Dead Letter Topics (DLT)

Simple Explanation: Dead Letter Topics (DLT) are special Kafka topics where failed messages are sent after all retry attempts are exhausted. They act as a "safety net" to prevent message loss and provide a mechanism for manual inspection and recovery of problematic messages.

Why Dead Letter Topics are Essential:

- **Message Preservation**: Ensure no messages are lost even after processing failures
- **Problem Isolation**: Separate failed messages from healthy processing flow
- **Manual Recovery**: Allow operators to inspect, fix, and reprocess failed messages
- **Debugging**: Provide visibility into failure patterns and root causes
- **System Reliability**: Prevent poison pills from blocking entire processing pipelines

Dead Letter Topic Architecture:

```
Dead Letter Topic Flow:

┌─────────────────────────────────────────────────────────────┐
│                    Normal Processing Flow                   │
│  Producer → Topic → Consumer → Business Logic → Success    │
└─────────────────────────────────────────────────────────────┘
                     │
                     │
                     ▼ (Exception)
┌─────────────────────────────────────────────────────────────┐
│                      Error Handler                         │
│  ┌─────────────────┐  ┌─────────────────┐               │
│  │ Retry Logic     │  │ Backoff Policy  │               │
│  │ (1st attempt)   │  │ (wait & retry)  │               │
│  └─────────────────┘  └─────────────────┘               │
└─────────────────────────────────────────────────────────────┘
                     │
                     │
                     ▼ (Retries Exhausted)
┌─────────────────────────────────────────────────────────────┐
│                 Dead Letter Publishing                      │
│  ┌─────────────────┐  ┌─────────────────┐               │
│  │ Add DLT Headers │  │ Route to DLT    │               │
│  │ (failure info)  │  │ (topic.DLT)     │               │
│  └─────────────────┘  └─────────────────┘               │
└─────────────────────────────────────────────────────────────┘
                     │
                     │
                     ▼
```

```
┌─────────────────────────────────────────────────────────────┐
│                    Dead Letter Topic                         │
│  Message + Headers: exception, stacktrace, retry count     │
│  ┌─────────────────┐  ┌─────────────────┐                 │
│  │ Manual Review   │  │ Fix & Republish │                 │
│  │ & Analysis      │  │ (if recoverable) │                 │
│  └─────────────────┘  └─────────────────┘                 │
└─────────────────────────────────────────────────────────────┘
```

Configuring DLT Publishing

Advanced Dead Letter Topic Configuration

```java
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.kafka.listener.DeadLetterPublishingRecoverer;
import org.springframework.kafka.listener.DefaultErrorHandler;
import org.springframework.kafka.core.KafkaTemplate;
import org.springframework.kafka.support.KafkaHeaders;

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

import java.util.function.BiFunction;

/**
 * Advanced Dead Letter Topic configuration and management
 */
@Configuration
@lombok.extern.slf4j.Slf4j
public class DeadLetterTopicConfiguration {

    @Autowired
    private KafkaTemplate<String, Object> kafkaTemplate;

    @Autowired
    private MeterRegistry meterRegistry;

    /**
     * Basic Dead Letter Topic configuration
     */
    @Bean("basicDltErrorHandler")
    public CommonErrorHandler basicDltErrorHandler() {

        // Simple DLT recoverer - sends to topic.DLT with same partition
        DeadLetterPublishingRecoverer recoverer = new
DeadLetterPublishingRecoverer(kafkaTemplate);

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

        // Configure retryable/non-retryable exceptions
        configureBasicExceptionClassification(errorHandler);

        log.info("Configured basic DLT error handler: 3 retries, 2s interval,
default DLT routing");

        return errorHandler;
    }

    /**
     * Advanced DLT configuration with custom routing and headers
     */
    @Bean("advancedDltErrorHandler")
    public CommonErrorHandler advancedDltErrorHandler() {

        // Custom destination resolver for intelligent DLT routing
        BiFunction<ConsumerRecord<?, ?>, Exception, TopicPartition>
destinationResolver =
            this::resolveDltDestination;

        DeadLetterPublishingRecoverer recoverer = new
DeadLetterPublishingRecoverer(
            kafkaTemplate, destinationResolver);

        // Enhanced header function to add diagnostic information
        recoverer.setHeadersFunction(this::enhanceDltHeaders);

        // Exponential backoff with max retries
        ExponentialBackOffWithMaxRetries backOff = new
ExponentialBackOffWithMaxRetries(5);
        backOff.setInitialInterval(1000L);
        backOff.setMultiplier(2.0);
        backOff.setMaxInterval(30000L);

        DefaultErrorHandler errorHandler = new DefaultErrorHandler(recoverer,
backOff);

        // Advanced exception classification
        configureAdvancedExceptionClassification(errorHandler);

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

        log.info("Configured advanced DLT error handler with custom routing and
enhanced headers");

        return errorHandler;
    }

    /**
```

```
     * Multi-tier DLT configuration for different failure types
     */
    @Bean("multiTierDltErrorHandler")
    public CommonErrorHandler multiTierDltErrorHandler() {

        // Multi-tier destination resolver
        BiFunction<ConsumerRecord<?, ?>, Exception, TopicPartition>
multiTierResolver =
            (record, exception) -> {
                String originalTopic = record.topic();
                int originalPartition = record.partition();

                // Determine DLT tier based on exception severity and type
                if (isPoisonPill(exception)) {
                    return new TopicPartition(originalTopic + ".poison.DLT",
originalPartition);

                } else if (isCriticalFailure(exception)) {
                    return new TopicPartition(originalTopic + ".critical.DLT",
originalPartition);

                } else if (isBusinessLogicFailure(exception)) {
                    return new TopicPartition(originalTopic + ".business.DLT",
originalPartition);

                } else if (isTransientFailure(exception)) {
                    return new TopicPartition(originalTopic + ".transient.DLT",
originalPartition);

                } else {
                    return new TopicPartition(originalTopic + ".unknown.DLT",
originalPartition);
                }
            };

        DeadLetterPublishingRecoverer recoverer = new
DeadLetterPublishingRecoverer(
            kafkaTemplate, multiTierResolver);

        // Add tier-specific headers
        recoverer.setHeadersFunction((record, exception) -> {
            Map<String, Object> headers = enhanceDltHeaders(record, exception);
            headers.put("dlt-tier", determineDltTier(exception));
            headers.put("severity", determineSeverity(exception));
            headers.put("recovery-strategy",
determineRecoveryStrategy(exception));
            return headers;
        });

        // Different backoff based on exception type
        DefaultErrorHandler errorHandler = new DefaultErrorHandler(recoverer);

        errorHandler.setBackOffFunction((record, ex) -> {
            if (isPoisonPill(ex)) {
                return new FixedBackOff(0L, 0L); // No retry for poison pills
            } else if (isCriticalFailure(ex)) {
                return new FixedBackOff(5000L, 2L); // Quick retry for critical
            } else if (isTransientFailure(ex)) {
                return new ExponentialBackOff(1000L, 2.0); // Exponential for
transient
            } else {
                return new FixedBackOff(2000L, 3L); // Default
            }
        });

        configureAdvancedExceptionClassification(errorHandler);

        log.info("Configured multi-tier DLT error handler with intelligent
routing");

        return errorHandler;
    }

    /**
     * Conditional DLT configuration - some messages go to retry, others to DLT
     */
    @Bean("conditionalDltErrorHandler")
    public CommonErrorHandler conditionalDltErrorHandler() {

        // Conditional recoverer
        ConsumerRecordRecoverer conditionalRecoverer = (record, exception) -> {

            // Check if message should be retried instead of going to DLT
            if (shouldRetryInsteadOfDlt(record, exception)) {
                sendToRetryTopic(record, exception);

            } else if (shouldQuarantine(record, exception)) {
                sendToQuarantineTopic(record, exception);

            } else {
                // Use standard DLT
                DeadLetterPublishingRecoverer standardRecoverer =
                    new DeadLetterPublishingRecoverer(kafkaTemplate);
                standardRecoverer.accept(record, exception);
            }
        };

        DefaultErrorHandler errorHandler = new
DefaultErrorHandler(conditionalRecoverer);

        // Very limited retries since we have conditional logic
        errorHandler.setBackOff(new FixedBackOff(1000L, 2L));

        configureAdvancedExceptionClassification(errorHandler);

        log.info("Configured conditional DLT error handler with retry/quarantine
logic");
```

```
        return errorHandler;
    }

    /**
     * Batched DLT configuration for high-volume scenarios
     */
    @Bean("batchedDltErrorHandler")
    public CommonErrorHandler batchedDltErrorHandler() {

        // Custom recoverer that batches DLT messages
        ConsumerRecordRecoverer batchedRecoverer = new
BatchedDltRecoverer(kafkaTemplate);

        DefaultErrorHandler errorHandler = new
DefaultErrorHandler(batchedRecoverer);

        // Shorter retries since we're batching
        errorHandler.setBackOff(new FixedBackOff(500L, 2L));

        configureAdvancedExceptionClassification(errorHandler);

        log.info("Configured batched DLT error handler for high-volume
processing");

        return errorHandler;
    }

    // Helper methods for DLT configuration
    private TopicPartition resolveDltDestination(ConsumerRecord<?, ?> record,
Exception exception) {
        String originalTopic = record.topic();
        int originalPartition = record.partition();

        // Custom routing logic based on message content and exception
        if (exception instanceof ValidationException) {
            return new TopicPartition(originalTopic + ".validation.DLT",
originalPartition);

        } else if (exception instanceof ExternalServiceException) {
            return new TopicPartition(originalTopic + ".external.DLT",
originalPartition);

        } else if (exception instanceof DatabaseException) {
            return new TopicPartition(originalTopic + ".database.DLT",
originalPartition);

        } else if (exception.getCause() instanceof
org.springframework.kafka.support.serializer.DeserializationException) {
            return new TopicPartition(originalTopic + ".poison.DLT",
originalPartition);

        } else {
            // Check message content for additional routing
            Object value = record.value();
            if (value instanceof OrderEvent) {
                return new TopicPartition(originalTopic + ".orders.DLT",
originalPartition);
            } else if (value instanceof PaymentEvent) {
                return new TopicPartition(originalTopic + ".payments.DLT",
originalPartition);
            } else {
                return new TopicPartition(originalTopic + ".DLT",
originalPartition);
            }
        }
    }

    private Map<String, Object> enhanceDltHeaders(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, exception.getMessage());

        // Enhanced diagnostic headers
        headers.put("dlt-timestamp", System.currentTimeMillis());
        headers.put("dlt-hostname", getHostname());
        headers.put("dlt-application", getApplicationName());
        headers.put("dlt-version", getApplicationVersion());
        headers.put("retry-count", getRetryCount(record));
        headers.put("processing-duration", calculateProcessingDuration(record));

        // Business context headers
        Object value = record.value();
        if (value instanceof OrderEvent order) {
            headers.put("business-order-id", order.getOrderId());
            headers.put("business-customer-id", order.getCustomerId());
            headers.put("business-amount", order.getAmount().toString());

        } else if (value instanceof PaymentEvent payment) {
            headers.put("business-payment-id", payment.getPaymentId());
            headers.put("business-payment-method", payment.getMethod());
            headers.put("business-amount", payment.getAmount().toString());
        }

        // Exception chain analysis
        Throwable cause = exception.getCause();
        if (cause != null) {
            headers.put(KafkaHeaders.DLT_EXCEPTION_CAUSE_FQCN,
cause.getClass().getName());
            headers.put("dlt-root-cause",
getRootCause(exception).getClass().getName());
```

```
        }

        // Stack trace (compressed for large stacks)
        String stackTrace = getStackTraceString(exception);
        if (stackTrace.length() > 4000) { // Kafka header limit consideration
            stackTrace = stackTrace.substring(0, 4000) + "...[truncated]";
        }
        headers.put(KafkaHeaders.DLT_EXCEPTION_STACKTRACE, stackTrace);

        return headers;
    }

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

                // Update retry metrics by exception type
                meterRegistry.counter("kafka.dlt.retry.attempts",
                    Tags.of(
                        "topic", record.topic(),
                        "exception", ex.getClass().getSimpleName(),
                        "attempt", String.valueOf(deliveryAttempt)
                    )).increment();
            }

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

                meterRegistry.counter("kafka.dlt.recovered",
                    Tags.of("topic", record.topic(), "exception",
ex.getClass().getSimpleName()))
                    .increment();
            }

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

                meterRegistry.counter("kafka.dlt.published",
                    Tags.of(
                        "topic", record.topic(),
                        "exception", original.getClass().getSimpleName(),
                        "dlt-destination", resolveDltDestination(record,
original).topic()
                    )).increment();
            }
        };
    }

    // Exception classification methods
    private boolean isPoisonPill(Exception exception) {
        return exception instanceof
org.springframework.kafka.support.serializer.DeserializationException ||
               exception instanceof
org.springframework.messaging.converter.MessageConversionException ||
               exception instanceof ClassCastException ||
               exception instanceof IllegalArgumentException;
    }

    private boolean isCriticalFailure(Exception exception) {
        return exception instanceof DatabaseException ||
               exception instanceof SecurityException ||
               (exception instanceof RuntimeException &&
                exception.getMessage() != null &&
                exception.getMessage().contains("CRITICAL"));
    }

    private boolean isBusinessLogicFailure(Exception exception) {
        return exception instanceof ValidationException ||
               exception instanceof BusinessRuleException ||
               exception instanceof WorkflowException;
    }

    private boolean isTransientFailure(Exception exception) {
        return exception instanceof ExternalServiceException ||
               exception instanceof java.util.concurrent.TimeoutException ||
               exception instanceof
org.springframework.dao.TransientDataAccessException ||
               exception instanceof java.net.ConnectException;
    }

    private String determineDltTier(Exception exception) {
        if (isPoisonPill(exception)) return "POISON";
        if (isCriticalFailure(exception)) return "CRITICAL";
        if (isBusinessLogicFailure(exception)) return "BUSINESS";
        if (isTransientFailure(exception)) return "TRANSIENT";
        return "UNKNOWN";
    }

    private String determineSeverity(Exception exception) {
        if (isCriticalFailure(exception)) return "HIGH";
        if (isBusinessLogicFailure(exception)) return "MEDIUM";
        if (isTransientFailure(exception)) return "LOW";
        return "MEDIUM";
    }
    private String determineRecoveryStrategy(Exception exception) {
        if (isPoisonPill(exception)) return "MANUAL_INSPECTION";
        if (isCriticalFailure(exception)) return "IMMEDIATE_ATTENTION";
        if (isBusinessLogicFailure(exception)) return "BUSINESS_REVIEW";
        if (isTransientFailure(exception)) return "RETRY_LATER";
        return "INVESTIGATE";
    }

    private boolean shouldRetryInsteadOfDlt(ConsumerRecord<?, ?> record, Exception
exception) {
        // Only retry transient failures and only for certain message types
        if (!isTransientFailure(exception)) {
            return false;
        }

        // Check retry count
        int retryCount = getRetryCount(record);
        if (retryCount >= 3) {
            return false; // Already retried enough
        }

        // Check message age
        long messageAge = System.currentTimeMillis() - record.timestamp();
        if (messageAge > Duration.ofHours(1).toMillis()) {
            return false; // Too old to retry
        }

        return true;
    }

    private boolean shouldQuarantine(ConsumerRecord<?, ?> record, Exception
exception) {
        return isPoisonPill(exception) ||
               (exception instanceof SecurityException) ||
               (exception.getMessage() != null &&
                exception.getMessage().contains("QUARANTINE"));
    }

    private void sendToRetryTopic(ConsumerRecord<?, ?> record, Exception
exception) {
        String retryTopic = record.topic() + ".retry";

        ProducerRecord<Object, Object> retryRecord = new ProducerRecord<>(
            retryTopic, record.partition(), record.key(), record.value());

        // Add retry headers
        retryRecord.headers().add("original-topic", record.topic().getBytes());
        retryRecord.headers().add("retry-count",
String.valueOf(getRetryCount(record) + 1).getBytes());
        retryRecord.headers().add("retry-reason",
exception.getMessage().getBytes());
        retryRecord.headers().add("retry-timestamp",
String.valueOf(System.currentTimeMillis()).getBytes());
```

```
        retryRecord.headers().add("scheduled-retry-time",
            String.valueOf(System.currentTimeMillis() +
Duration.ofMinutes(5).toMillis()).getBytes());

        kafkaTemplate.send(retryRecord);

        log.info("Sent to retry topic: original={}, retry={}, attempt={}",
            record.topic(), retryTopic, getRetryCount(record) + 1);
    }

    private void sendToQuarantineTopic(ConsumerRecord<?, ?> record, Exception
exception) {
        String quarantineTopic = record.topic() + ".quarantine";

        ProducerRecord<Object, Object> quarantineRecord = new ProducerRecord<>(
            quarantineTopic, record.partition(), record.key(), record.value());

        // Add quarantine headers
        quarantineRecord.headers().add("quarantine-reason",
determineDltTier(exception).getBytes());
        quarantineRecord.headers().add("quarantine-timestamp",
            String.valueOf(System.currentTimeMillis()).getBytes());
        quarantineRecord.headers().add("requires-manual-review",
"true".getBytes());

        kafkaTemplate.send(quarantineRecord);

        log.warn("Sent to quarantine topic: original={}, quarantine={}, reason=
{}",
            record.topic(), quarantineTopic, determineDltTier(exception));
    }

    // Utility methods
    private void configureBasicExceptionClassification(DefaultErrorHandler
errorHandler) {
        errorHandler.addNotRetryableExceptions(
            IllegalArgumentException.class,
            ClassCastException.class,

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

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

    private void configureAdvancedExceptionClassification(DefaultErrorHandler
errorHandler) {
        // Fatal exceptions - immediate DLT
        errorHandler.addNotRetryableExceptions(
            IllegalArgumentException.class,
            NullPointerException.class,
            ClassCastException.class,

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

org.springframework.messaging.converter.MessageConversionException.class,
            SecurityException.class
        );

        // Retryable exceptions
        errorHandler.addRetryableExceptions(
            ValidationException.class,
            ExternalServiceException.class,
            DatabaseException.class,
            java.util.concurrent.TimeoutException.class,
            org.springframework.dao.TransientDataAccessException.class,
            java.net.ConnectException.class
        );

        // Reset retry state on exception type change
        errorHandler.setResetStateOnExceptionChange(true);
    }

    private int getRetryCount(ConsumerRecord<?, ?> record) {
        Header retryHeader = record.headers().lastHeader("retry-count");
        if (retryHeader != null) {
            return Integer.parseInt(new String(retryHeader.value()));
        }
        return 0;
    }

    private long calculateProcessingDuration(ConsumerRecord<?, ?> record) {
        Header startHeader = record.headers().lastHeader("processing-start");
        if (startHeader != null) {
            long startTime = Long.parseLong(new String(startHeader.value()));
            return System.currentTimeMillis() - startTime;
        }
        return 0;
    }

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

    private String getApplicationName() {
        return "kafka-consumer-app"; // Could read from properties
    }

    private String getApplicationVersion() {
        return "1.0.0"; // Could read from manifest
    }
    private Throwable getRootCause(Throwable throwable) {
        Throwable cause = throwable;
        while (cause.getCause() != null) {
            cause = cause.getCause();
        }
        return cause;
    }

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

/**
 * Batched DLT recoverer for high-volume scenarios
 */
@Component
@lombok.extern.slf4j.Slf4j
public class BatchedDltRecoverer implements ConsumerRecordRecoverer {

    private final KafkaTemplate<String, Object> kafkaTemplate;
    private final Map<String, List<ProducerRecord<Object, Object>>> batchedRecords
= new ConcurrentHashMap<>();
    private final ScheduledExecutorService scheduler =
Executors.newScheduledThreadPool(2);

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

        // Schedule batch flushing every 5 seconds
        scheduler.scheduleAtFixedRate(this::flushBatches, 5, 5, TimeUnit.SECONDS);

        // Schedule batch flushing when batch size reaches threshold
        scheduler.scheduleAtFixedRate(this::flushLargeBatches, 1, 1,
TimeUnit.SECONDS);
    }

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

        ProducerRecord<Object, Object> dltRecord = new ProducerRecord<>(
            dltTopic, record.partition(), record.key(), record.value());

        // Add DLT headers
        dltRecord.headers().add("dlt-batch-timestamp",
String.valueOf(System.currentTimeMillis()).getBytes());
        dltRecord.headers().add("dlt-exception",
exception.getClass().getSimpleName().getBytes());
```

```
        // Add to batch
        batchedRecords.computeIfAbsent(dltTopic, k -> new ArrayList<>
()).add(dltRecord);

        log.debug("Added record to DLT batch: topic={}, batchSize={}",
            dltTopic, batchedRecords.get(dltTopic).size());
    }

    private void flushBatches() {
        batchedRecords.forEach((topic, records) -> {
            if (!records.isEmpty()) {
                log.info("Flushing DLT batch: topic={}, size={}", topic,
records.size());

                records.forEach(kafkaTemplate::send);
                records.clear();
            }
        });
    }

    private void flushLargeBatches() {
        batchedRecords.forEach((topic, records) -> {
            if (records.size() >= 100) { // Threshold for large batches
                log.info("Flushing large DLT batch: topic={}, size={}", topic,
records.size());

                records.forEach(kafkaTemplate::send);
                records.clear();
            }
        });
    }

    @PreDestroy
    public void shutdown() {
        log.info("Shutting down batched DLT recoverer");
        flushBatches(); // Final flush
        scheduler.shutdown();
    }
}

// Custom exception classes for demonstration
class BusinessRuleException extends Exception {
    public BusinessRuleException(String message) { super(message); }
}

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

Retrying from DLT

DLT Processing and Retry Patterns

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

/**
 * Dead Letter Topic processing and retry patterns
 */
@Component
@lombok.extern.slf4j.Slf4j
public class DeadLetterTopicProcessor {

    @Autowired
    private KafkaTemplate<String, Object> kafkaTemplate;

    @Autowired
    private DltAnalysisService dltAnalysisService;

    @Autowired
    private MessageRepairService messageRepairService;

    /**
     * Manual DLT processing - inspect and potentially reprocess messages
     */
    @KafkaListener(
        topics = "orders.DLT",
        groupId = "dlt-manual-processor"
    )
    public void processOrdersDlt(
            @Payload OrderEvent order,
            @Header(KafkaHeaders.RECEIVED_TOPIC) String dltTopic,
            @Header(name = KafkaHeaders.DLT_ORIGINAL_TOPIC, required = false)
String originalTopic,
            @Header(name = KafkaHeaders.DLT_EXCEPTION_FQCN, required = false)
String exceptionClass,
            @Header(name = KafkaHeaders.DLT_EXCEPTION_MESSAGE, required = false)
String exceptionMessage,
            @Header(name = "retry-count", required = false) String
retryCountHeader,
            ConsumerRecord<String, OrderEvent> record) {

        log.info("Processing DLT message: orderId={}, originalTopic={}, exception=
{}",
            order.getOrderId(), originalTopic, exceptionClass);

        try {
            // Analyze the failure
            DltAnalysisResult analysis = dltAnalysisService.analyzeDltMessage(
                record, exceptionClass, exceptionMessage);

            if (analysis.isRecoverable()) {
                log.info("DLT message is recoverable: orderId={}, strategy={}",
                    order.getOrderId(), analysis.getRecoveryStrategy());

                // Attempt to fix and reprocess
                handleRecoverableDltMessage(order, analysis, originalTopic);

            } else {
                log.warn("DLT message is not recoverable: orderId={}, reason={}",
                    order.getOrderId(), analysis.getFailureReason());

                // Log for manual intervention
                handleNonRecoverableDltMessage(order, analysis);
            }

        } catch (Exception e) {
            log.error("Failed to process DLT message: orderId={}",
order.getOrderId(), e);

            // Send to manual review queue
            sendToManualReview(record, e);
        }
    }

    /**
     * Automated DLT retry processor
     */
    @KafkaListener(
        topics = {"orders.transient.DLT", "payments.transient.DLT"},
        groupId = "dlt-auto-retry-processor"
    )
    public void processTransientDlt(
            @Payload Object message,
            @Header(KafkaHeaders.RECEIVED_TOPIC) String dltTopic,
            @Header(name = KafkaHeaders.DLT_ORIGINAL_TOPIC) String originalTopic,
            @Header(name = "dlt-timestamp") long dltTimestamp,
            ConsumerRecord<String, Object> record) {

        // Only retry if message is not too old
        long messageAge = System.currentTimeMillis() - dltTimestamp;
        if (messageAge > Duration.ofHours(24).toMillis()) {
            log.info("DLT message too old, skipping retry: age={}h, topic={}",
                messageAge / 3600000, dltTopic);

            sendToExpiredDlt(record);
            return;
        }

        log.info("Attempting automated retry for transient DLT: originalTopic={},
age={}min",
            originalTopic, messageAge / 60000);

        try {
            // Check if the transient issue might be resolved
            if (isSystemHealthy() && canRetryMessage(message)) {
                // Add retry tracking headers
                ProducerRecord<Object, Object> retryRecord = new ProducerRecord<>(
                    originalTopic, record.key(), message);

                retryRecord.headers().add("dlt-retry", "true".getBytes());
                retryRecord.headers().add("dlt-retry-timestamp",
                    String.valueOf(System.currentTimeMillis()).getBytes());
                retryRecord.headers().add("original-dlt-topic",
dltTopic.getBytes());

                kafkaTemplate.send(retryRecord);

                log.info("Successfully sent DLT message back to original topic: {}
-> {}",
                    dltTopic, originalTopic);

            } else {
                log.info("System not ready for DLT retry, will try later: topic=
{}", dltTopic);

                // Send back to DLT with delay
                sendToDltWithDelay(record, Duration.ofMinutes(30));
            }

        } catch (Exception e) {
            log.error("Failed to process transient DLT message", e);

            // Move to permanent DLT
            sendToPermanentDlt(record, e);
        }
    }

    /**
     * DLT handler using @DltHandler annotation (Spring Kafka 2.5+)
     */
    @Component
    public static class AnnotationBasedDltHandler {

        @Autowired
        private DltRepairService repairService;

        /**
         * Primary message listener
         */
        @KafkaListener(
            topics = "annotated-orders",
            groupId = "annotated-order-processor"
        )
        public void processOrder(OrderEvent order) {
            log.info("Processing order: {}", order.getOrderId());

            // Simulate processing that might fail
            if (order.getOrderId().contains("FAIL")) {
                throw new ValidationException("Simulated validation failure");
            }

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

        /**
         * DLT handler for the same listener group
         */
        @DltHandler
        public void handleDltOrder(
                OrderEvent order,
                @Header(KafkaHeaders.DLT_EXCEPTION_MESSAGE) String
exceptionMessage,
                @Header(KafkaHeaders.DLT_EXCEPTION_FQCN) String exceptionClass) {

            log.warn("Handling DLT order: orderId={}, exception={}, message={}",
                order.getOrderId(), exceptionClass, exceptionMessage);

            try {
                // Attempt repair based on exception type
                if (exceptionClass.contains("ValidationException")) {
                    OrderEvent repairedOrder =
repairService.repairValidation(order);
                    if (repairedOrder != null) {
                        log.info("Repaired and reprocessing order: {}",
order.getOrderId());
                        processOrder(repairedOrder);
                        return;
                    }
                }

                // If repair fails, log for manual intervention
                log.error("Could not repair DLT order, requires manual
intervention: {}",
                    order.getOrderId());

            } catch (Exception e) {
                log.error("DLT handler failed: orderId={}", order.getOrderId(),
e);
            }
        }
    }

    /**
     * Scheduled DLT reprocessing job
     */
    @Scheduled(fixedDelay = 3600000) // Every hour
    public void scheduledDltReprocessing() {
        log.info("Starting scheduled DLT reprocessing job");

        try {
            // Get list of DLT topics to process
            List<String> dltTopics = getDltTopicsForReprocessing();
            for (String dltTopic : dltTopics) {
                processDltTopicForRetry(dltTopic);
            }

        } catch (Exception e) {
            log.error("Scheduled DLT reprocessing failed", e);
        }
    }

    /**
     * Batch DLT processing for efficiency
     */
    @KafkaListener(
        topics = "high-volume.DLT",
        groupId = "dlt-batch-processor",
        containerFactory = "batchListenerContainerFactory"
    )
    public void processDltBatch(List<ConsumerRecord<String, Object>> records) {
        log.info("Processing DLT batch: size={}", records.size());

        Map<String, List<ConsumerRecord<String, Object>>> groupedByException =
records.stream()
            .collect(Collectors.groupingBy(record -> {
                Header exceptionHeader =
record.headers().lastHeader(KafkaHeaders.DLT_EXCEPTION_FQCN);
                return exceptionHeader != null ? new
String(exceptionHeader.value()) : "Unknown";
            }));

        // Process each exception type separately
        groupedByException.forEach(this::processDltByExceptionType);
    }

    // Helper methods for DLT processing
    private void handleRecoverableDltMessage(OrderEvent order, DltAnalysisResult
analysis, String originalTopic) {
        try {
            switch (analysis.getRecoveryStrategy()) {
                case "FIX_AND_RETRY" -> {
                    OrderEvent fixedOrder =
messageRepairService.repairOrder(order, analysis);
                    if (fixedOrder != null) {
                        republishToOriginalTopic(fixedOrder, originalTopic);
                    }
                }
                case "RETRY_AFTER_DELAY" -> {
                    scheduleDelayedRetry(order, originalTopic,
Duration.ofMinutes(30));
                }
                case "MANUAL_REVIEW" -> {
                    sendToManualReviewQueue(order, analysis);
                }
                default -> {
                    log.warn("Unknown recovery strategy: {}",
analysis.getRecoveryStrategy());
                }
            }
        } catch (Exception e) {
            log.error("Recovery failed for order: {}", order.getOrderId(), e);
        }
    }

    private void handleNonRecoverableDltMessage(OrderEvent order,
DltAnalysisResult analysis) {
        // Log to metrics and alerting systems
        meterRegistry.counter("kafka.dlt.nonrecoverable",
            Tags.of("reason", analysis.getFailureReason()))
            .increment();

        // Store in database for audit trail
        dltAnalysisService.recordNonRecoverableFailure(order, analysis);

        // Alert operations team if needed
        if (analysis.requiresImmedateAttention()) {
            alertOperationsTeam(order, analysis);
        }
    }

    private void sendToManualReview(ConsumerRecord<String, OrderEvent> record,
Exception e) {
        ManualReviewTask reviewTask = ManualReviewTask.builder()
            .originalTopic(record.topic())
            .messageKey(record.key())
            .messageValue(record.value())
            .processingException(e.getMessage())
            .timestamp(System.currentTimeMillis())
            .priority("HIGH")
            .build();

        kafkaTemplate.send("manual-review-queue", reviewTask);
    }

    private boolean isSystemHealthy() {
        // Check system health indicators
        return true; // Placeholder
    }

    private boolean canRetryMessage(Object message) {
        // Business logic to determine if message can be retried
        return true; // Placeholder
    }

    private void sendToExpiredDlt(ConsumerRecord<String, Object> record) {
        String expiredDltTopic = record.topic().replace(".DLT", ".expired.DLT");
        kafkaTemplate.send(expiredDltTopic, record.key(), record.value());
    }
    private void sendToDltWithDelay(ConsumerRecord<String, Object> record,
Duration delay) {
        ProducerRecord<Object, Object> delayedRecord = new ProducerRecord<>(
            record.topic(), record.key(), record.value());

        delayedRecord.headers().add("retry-after",
            String.valueOf(System.currentTimeMillis() +
delay.toMillis()).getBytes());

        kafkaTemplate.send(delayedRecord);
    }

    private void sendToPermanentDlt(ConsumerRecord<String, Object> record,
Exception e) {
        String permanentDltTopic = record.topic().replace(".transient.DLT",
".permanent.DLT");

        ProducerRecord<Object, Object> permanentRecord = new ProducerRecord<>(
            permanentDltTopic, record.key(), record.value());

        permanentRecord.headers().add("permanent-failure-reason",
e.getMessage().getBytes());
        permanentRecord.headers().add("permanent-failure-timestamp",
            String.valueOf(System.currentTimeMillis()).getBytes());

        kafkaTemplate.send(permanentRecord);
    }

    private void republishToOriginalTopic(OrderEvent order, String originalTopic)
{
        ProducerRecord<Object, Object> republishRecord = new ProducerRecord<>(
            originalTopic, order.getOrderId(), order);

        republishRecord.headers().add("dlt-recovered", "true".getBytes());
        republishRecord.headers().add("recovery-timestamp",
            String.valueOf(System.currentTimeMillis()).getBytes());

        kafkaTemplate.send(republishRecord);

        log.info("Republished recovered message: orderId={}, topic={}",
            order.getOrderId(), originalTopic);
    }

    private void scheduleDelayedRetry(OrderEvent order, String originalTopic,
Duration delay) {
        // Implementation would use a scheduler or delayed message system
        log.info("Scheduled delayed retry: orderId={}, delay={}min",
            order.getOrderId(), delay.toMinutes());
    }

    private void sendToManualReviewQueue(OrderEvent order, DltAnalysisResult
analysis) {
        ManualReviewTask task = ManualReviewTask.builder()
            .orderId(order.getOrderId())
```

```
                .analysisResult(analysis)
                .priority(analysis.requiresImmedateAttention() ? "HIGH" : "MEDIUM")
                .build();

            kafkaTemplate.send("manual-review-tasks", task);
        }

        private List<String> getDltTopicsForReprocessing() {
            // Implementation would query Kafka admin client for DLT topics
            return Arrays.asList("orders.transient.DLT", "payments.transient.DLT");
        }

        private void processDltTopicForRetry(String dltTopic) {
            log.info("Processing DLT topic for retry: {}", dltTopic);
            // Implementation would consume from DLT topic and apply retry logic
        }

        private void processDltByExceptionType(String exceptionType,
                                               List<ConsumerRecord<String, Object>>
    records) {
            log.info("Processing DLT batch by exception type: type={}, count={}",
                exceptionType, records.size());

            // Apply exception-specific recovery strategies
            for (ConsumerRecord<String, Object> record : records) {
                try {
                    // Process based on exception type
                    processRecordByExceptionType(record, exceptionType);
                } catch (Exception e) {
                    log.error("Failed to process DLT record: topic={}, offset={}",
                        record.topic(), record.offset(), e);
                }
            }
        }

        private void processRecordByExceptionType(ConsumerRecord<String, Object>
    record, String exceptionType) {
            switch (exceptionType) {
                case "ValidationException" -> attemptValidationFix(record);
                case "ExternalServiceException" -> attemptServiceRetry(record);
                case "DatabaseException" -> attemptDatabaseRetry(record);
                default -> logUnknownExceptionType(record, exceptionType);
            }
        }

        private void attemptValidationFix(ConsumerRecord<String, Object> record) {
            // Attempt to fix validation issues
            log.debug("Attempting validation fix for record: topic={}, offset={}",
                record.topic(), record.offset());
        }

        private void attemptServiceRetry(ConsumerRecord<String, Object> record) {
            // Retry external service calls
            log.debug("Attempting service retry for record: topic={}, offset={}",
                record.topic(), record.offset());
        }

        private void attemptDatabaseRetry(ConsumerRecord<String, Object> record) {
            // Retry database operations
            log.debug("Attempting database retry for record: topic={}, offset={}",
                record.topic(), record.offset());
        }

        private void logUnknownExceptionType(ConsumerRecord<String, Object> record,
    String exceptionType) {
            log.warn("Unknown exception type in DLT: type={}, topic={}, offset={}",
                exceptionType, record.topic(), record.offset());
        }

        private void alertOperationsTeam(OrderEvent order, DltAnalysisResult analysis)
    {
            log.error("🚨 ALERT: Non-recoverable DLT message requires immediate
    attention: orderId={}, reason={}",
                order.getOrderId(), analysis.getFailureReason());
            // Integration with alerting systems (PagerDuty, Slack, etc.)
        }

        @Autowired
        private MeterRegistry meterRegistry;
    }

    // Supporting classes for DLT processing
    @Service
    public class DltAnalysisService {

        public DltAnalysisResult analyzeDltMessage(ConsumerRecord<String, OrderEvent>
    record,
                                                 String exceptionClass, String
    exceptionMessage) {

            DltAnalysisResult.DltAnalysisResultBuilder builder =
    DltAnalysisResult.builder()
                .messageId(record.key())
                .exceptionType(exceptionClass)
                .exceptionMessage(exceptionMessage);

            // Analyze based on exception type
            if ("ValidationException".equals(exceptionClass)) {
                return builder
                    .recoverable(true)
                    .recoveryStrategy("FIX_AND_RETRY")
                    .confidence(0.8)
                    .build();

            } else if ("ExternalServiceException".equals(exceptionClass)) {
                return builder
                    .recoverable(true)
                    .recoveryStrategy("RETRY_AFTER_DELAY")
```

```java
                .confidence(0.9)
                .build();

        } else if ("DeserializationException".equals(exceptionClass)) {
            return builder
                .recoverable(false)
                .failureReason("POISON_PILL")
                .requiresImmedateAttention(true)
                .build();

        } else {
            return builder
                .recoverable(false)
                .failureReason("UNKNOWN_EXCEPTION")
                .requiresImmedateAttention(false)
                .build();
        }
    }

    public void recordNonRecoverableFailure(OrderEvent order, DltAnalysisResult
analysis) {
        log.info("Recording non-recoverable failure: orderId={}, reason={}",
            order.getOrderId(), analysis.getFailureReason());
        // Store in database or metrics system
    }
}

@Service
public class MessageRepairService {

    public OrderEvent repairOrder(OrderEvent order, DltAnalysisResult analysis) {
        try {
            OrderEvent.OrderEventBuilder builder = order.toBuilder();

            // Apply repairs based on analysis
            if ("ValidationException".equals(analysis.getExceptionType())) {
                // Fix common validation issues
                if (order.getCustomerId() == null) {
                    builder.customerId("UNKNOWN_CUSTOMER");
                }

                if (order.getAmount() == null) {
                    builder.amount(java.math.BigDecimal.ZERO);
                }

                if (order.getStatus() == null) {
                    builder.status("PENDING_REPAIR");
                }
            }

            OrderEvent repaired = builder.build();
            log.info("Repaired order: original={}, repaired={}", order, repaired);

            return repaired;
        } catch (Exception e) {
            log.error("Failed to repair order: {}", order.getOrderId(), e);
            return null;
        }
    }
}

@Service
public class DltRepairService {

    public OrderEvent repairValidation(OrderEvent order) {
        // Implement validation repair logic
        return order.toBuilder()
            .status("REPAIRED")
            .build();
    }
}

// Supporting data structures
@lombok.Data
@lombok.Builder
@lombok.NoArgsConstructor
@lombok.AllArgsConstructor
class DltAnalysisResult {
    private String messageId;
    private String exceptionType;
    private String exceptionMessage;
    private boolean recoverable;
    private String recoveryStrategy;
    private String failureReason;
    private double confidence;
    private boolean requiresImmedateAttention;
}

@lombok.Data
@lombok.Builder
@lombok.NoArgsConstructor
@lombok.AllArgsConstructor
class ManualReviewTask {
    private String originalTopic;
    private String messageKey;
    private Object messageValue;
    private String processingException;
    private long timestamp;
    private String priority;
    private String orderId;
    private DltAnalysisResult analysisResult;
}
```

This comprehensive section covers Dead Letter Topics (DLT) configuration, publishing, and retry patterns with extensive Java examples for production scenarios. The guide continues with the legacy

SeekToCurrentErrorHandler section next.