Spring Kafka Error Handling & Retry: Complete Developer Guide

A comprehensive guide covering all aspects of Spring Kafka error handling and retry mechanisms, from modern DefaultErrorHandler to legacy SeekToCurrentErrorHandler with extensive Java examples and production patterns.

Table of Contents

What is Error Handling in Kafka?

Simple Explanation: Error handling in Kafka determines what happens when message processing fails. Spring Kafka provides sophisticated mechanisms to retry failed messages, send them to dead letter topics, or recover gracefully, ensuring your application remains resilient in the face of processing failures.

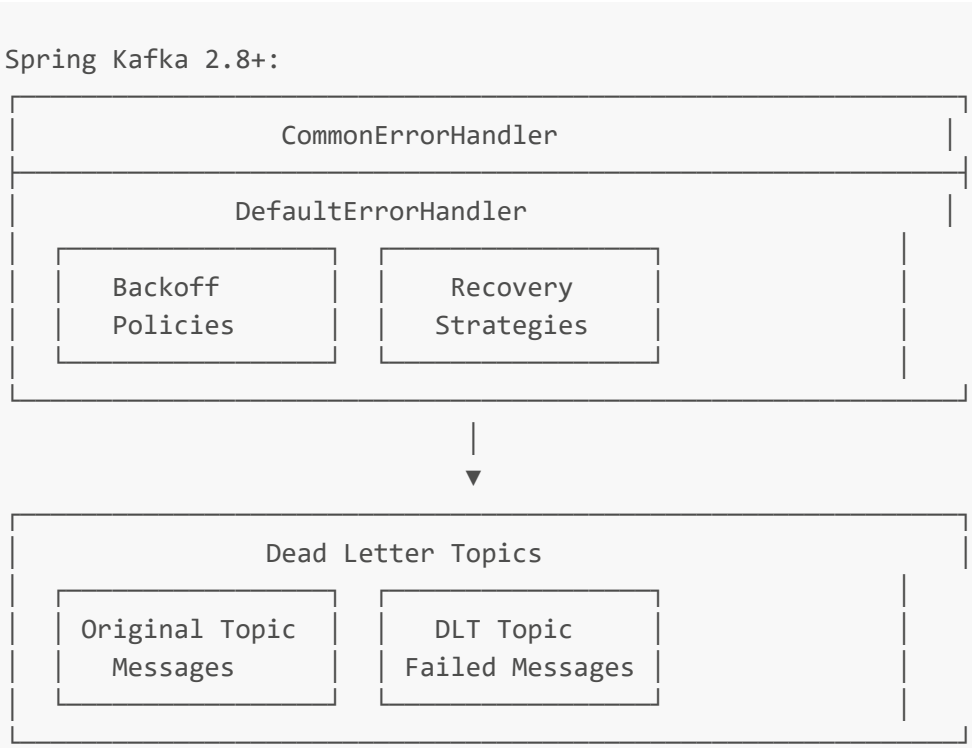
Why Error Handling is Critical:

- **Resilience**: Handle transient failures without losing messages
- **Reliability**: Ensure critical messages are eventually processed
- **Monitoring**: Track and alert on processing failures
- **Recovery**: Provide mechanisms to reprocess failed messages
- **System Stability**: Prevent cascading failures from propagating

Spring Kafka Error Handling Evolution:

```
Error Handling Architecture Evolution:

Spring Kafka 2.7 and Earlier:
┌─────────────────────────────────────────────────────────┐
│                Legacy Error Handlers                    │
├─────────────────────────────────────────────────────────┤
│ ErrorHandler (Record)    │ BatchErrorHandler (Batch)    │
│ SeekToCurrentErrorHandler │ RecoveringBatchErrorHandler  │
└─────────────────────────────────────────────────────────┘
```



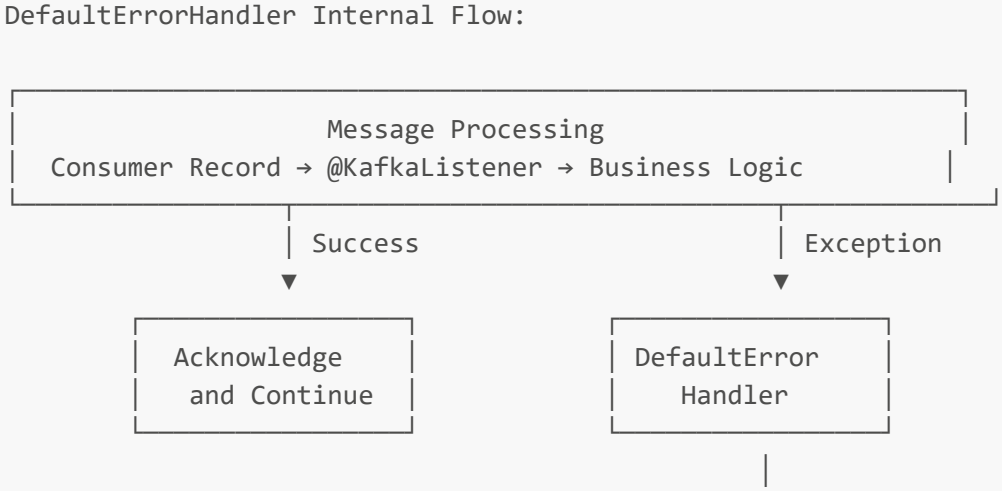
🔧 DefaultErrorHandler (Spring Kafka 2.8+)

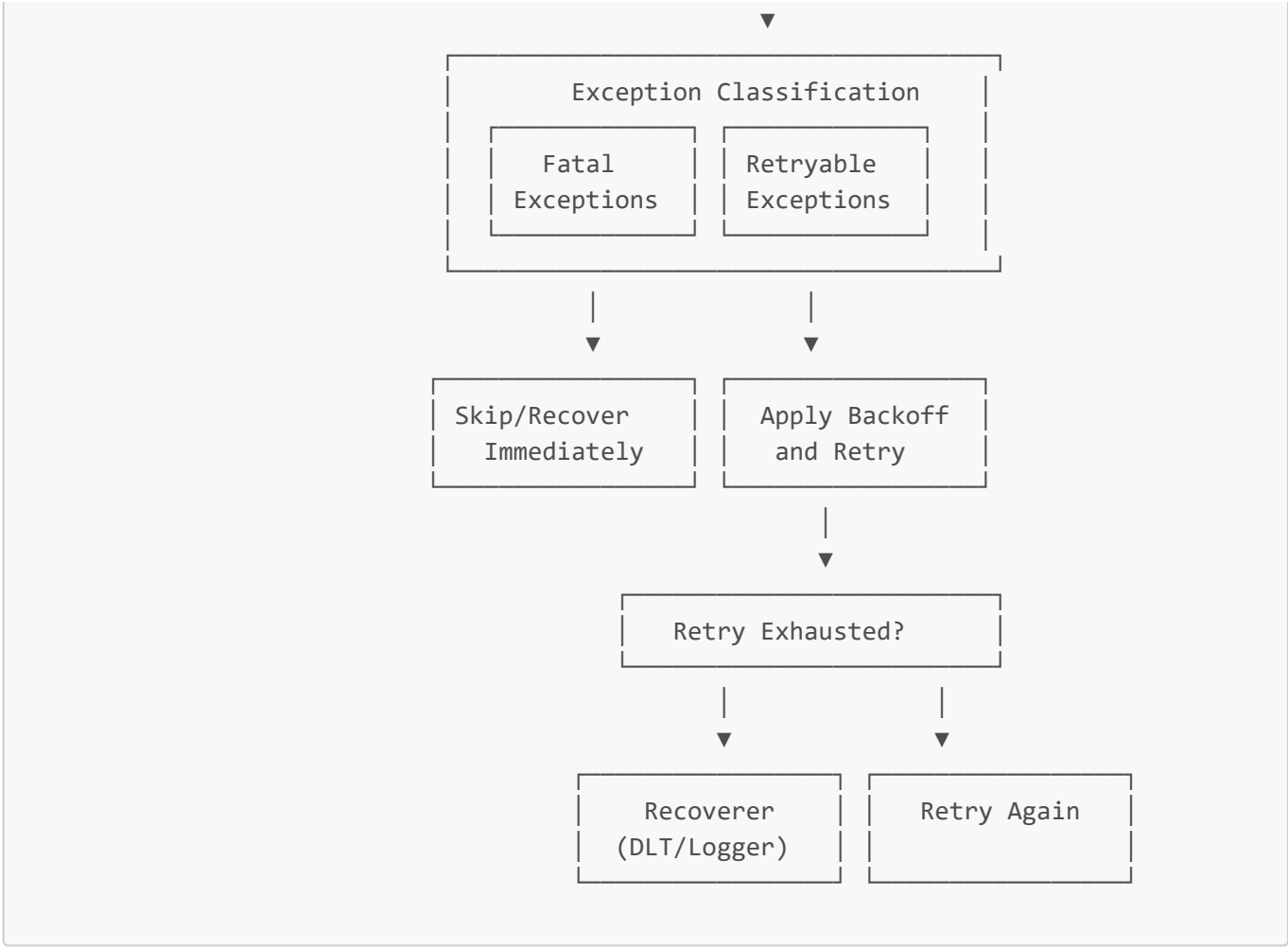
Simple Explanation: DefaultErrorHandler is the modern, unified error handler in Spring Kafka that replaced multiple legacy handlers. It provides comprehensive retry mechanisms, backoff policies, and recovery strategies for both record and batch listeners.

Why DefaultErrorHandler Exists:

- **Unification**: Single handler for both record and batch listeners
- **Flexibility**: Configurable retry policies and recovery strategies
- **Performance**: Non-blocking retries and optimized recovery
- **Observability**: Better metrics and monitoring capabilities
- **Simplicity**: Easier configuration and management

Internal DefaultErrorHandler Architecture:





Backoff Policies (Fixed, Exponential)

Advanced Backoff Configuration and Patterns

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

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

import java.time.Duration;
import java.util.function.BiFunction;

/**
 * Advanced DefaultErrorHandler configuration with comprehensive backoff policies
 */
@Configuration
@lombok.extern.slf4j.Slf4j
```

```
public class ErrorHandlerConfiguration {

    @Autowired
    private KafkaTemplate<String, Object> kafkaTemplate;

    /**
     * Fixed backoff error handler - predictable retry intervals
     */
    @Bean("fixedBackoffErrorHandler")
    public CommonErrorHandler fixedBackoffErrorHandler() {
        // Fixed backoff: 2 seconds between retries, maximum 5 attempts
        FixedBackOff fixedBackOff = new FixedBackOff(2000L, 5L);

        DefaultErrorHandler errorHandler = new DefaultErrorHandler(fixedBackOff);

        // Configure exception classification
        configureExceptionClassification(errorHandler);

        // Add retry listeners for monitoring
        errorHandler.setRetryListeners(createRetryListener("FixedBackoff"));

        log.info("Configured FixedBackoff error handler: interval=2s,
maxAttempts=5");

        return errorHandler;
    }

    /**
     * Exponential backoff error handler - increasing retry intervals
     */
    @Bean("exponentialBackoffErrorHandler")
    public CommonErrorHandler exponentialBackoffErrorHandler() {
        // Exponential backoff: starts at 1s, multiplier 2.0, max 30s, max 10
minutes total
        ExponentialBackOff exponentialBackOff = new ExponentialBackOff();
        exponentialBackOff.setInitialInterval(1000L);        // Start with 1
second
        exponentialBackOff.setMultiplier(2.0);               // Double each time
        exponentialBackOff.setMaxInterval(30000L);           // Cap at 30 seconds
        exponentialBackOff.setMaxElapsedTime(600000L);       // Stop after 10
minutes total

        DefaultErrorHandler errorHandler = new
DefaultErrorHandler(exponentialBackOff);

        configureExceptionClassification(errorHandler);
        errorHandler.setRetryListeners(createRetryListener("ExponentialBackoff"));

        log.info("Configured ExponentialBackoff error handler: initial=1s,
multiplier=2.0, max=30s");

        return errorHandler;
    }
```

```
    /**
     * Exponential backoff with max retries - convenient wrapper
     */
    @Bean("exponentialWithMaxRetriesHandler")
    public CommonErrorHandler exponentialWithMaxRetriesHandler() {
        // Exponential backoff with max retries: 6 attempts total
        ExponentialBackOffWithMaxRetries backOff = new
ExponentialBackOffWithMaxRetries(6);
        backOff.setInitialInterval(1000L);     // 1 second
        backOff.setMultiplier(2.0);            // Double each time
        backOff.setMaxInterval(10000L);        // Cap at 10 seconds

        // This will retry after: 1s, 2s, 4s, 8s, 10s, 10s then recover

        DefaultErrorHandler errorHandler = new DefaultErrorHandler(backOff);

        configureExceptionClassification(errorHandler);

errorHandler.setRetryListeners(createRetryListener("ExponentialWithMaxRetries"));

        log.info("Configured ExponentialBackOffWithMaxRetries: maxRetries=6,
intervals=[1s,2s,4s,8s,10s,10s]");

        return errorHandler;
    }

    /**
     * Custom backoff policy based on exception type
     */
    @Bean("customBackoffErrorHandler")
    public CommonErrorHandler customBackoffErrorHandler() {
        DefaultErrorHandler errorHandler = new DefaultErrorHandler();

        // Custom backoff function based on record and exception
        BiFunction<ConsumerRecord<?, ?>, Exception, BackOff> backoffFunction =
            (record, exception) -> {

                if (exception instanceof ValidationException) {
                    // Fast retry for validation errors (likely transient)
                    return new FixedBackOff(500L, 3L);

                } else if (exception instanceof ExternalServiceException) {
                    // Exponential backoff for external service errors
                    ExponentialBackOff exponentialBackOff = new
ExponentialBackOff();
                    exponentialBackOff.setInitialInterval(2000L);
                    exponentialBackOff.setMultiplier(1.5);
                    exponentialBackOff.setMaxInterval(60000L);
                    exponentialBackOff.setMaxElapsedTime(300000L); // 5 minutes
                    return exponentialBackOff;

                } else if (exception instanceof DatabaseException) {
                    // Longer fixed backoff for database errors
                    return new FixedBackOff(5000L, 10L);
```

```
                } else {
                    // Default exponential backoff for unknown errors
                    return new ExponentialBackOff(1000L, 2.0);
                }
            };

        errorHandler.setBackOffFunction(backoffFunction);

        configureExceptionClassification(errorHandler);
        errorHandler.setRetryListeners(createRetryListener("CustomBackoff"));

        log.info("Configured CustomBackoff error handler with exception-specific
policies");

        return errorHandler;
    }

    /**
     * Jittered exponential backoff to prevent thundering herd
     */
    @Bean("jitteredBackoffErrorHandler")
    public CommonErrorHandler jitteredBackoffErrorHandler() {

        // Custom jittered exponential backoff
        BackOff jitteredBackOff = new BackOff() {
            private final ExponentialBackOff baseBackOff = new
ExponentialBackOff(1000L, 2.0);
            private final Random random = new Random();

            @Override
            public BackOffExecution start() {
                return new BackOffExecution() {
                    private final BackOffExecution baseExecution =
baseBackOff.start();

                    @Override
                    public long nextBackOff() {
                        long baseInterval = baseExecution.nextBackOff();
                        if (baseInterval == BackOffExecution.STOP) {
                            return BackOffExecution.STOP;
                        }

                        // Add ±25% jitter to prevent synchronized retries
                        double jitterFactor = 0.75 + (random.nextDouble() * 0.5);
// 0.75 to 1.25
                        long jitteredInterval = (long) (baseInterval *
jitterFactor);

                        log.debug("Jittered backoff: base={}ms, jittered={}ms",
baseInterval, jitteredInterval);
                        return jitteredInterval;
                    }
                };
```

```
            }
        };

        DefaultErrorHandler errorHandler = new
DefaultErrorHandler(jitteredBackOff);

        configureExceptionClassification(errorHandler);
        errorHandler.setRetryListeners(createRetryListener("JitteredBackoff"));

        log.info("Configured JitteredBackoff error handler with ±25%
randomization");

        return errorHandler;
    }

    /**
     * Adaptive backoff based on system load
     */
    @Bean("adaptiveBackoffErrorHandler")
    public CommonErrorHandler adaptiveBackoffErrorHandler() {

        DefaultErrorHandler errorHandler = new DefaultErrorHandler();

        // Adaptive backoff function based on system metrics
        BiFunction<ConsumerRecord<?, ?>, Exception, BackOff> adaptiveFunction =
            (record, exception) -> {

                // Get current system metrics
                double cpuUsage = getCurrentCpuUsage();
                int queueSize = getCurrentQueueSize();

                if (cpuUsage > 0.8 || queueSize > 1000) {
                    // System under stress - use longer backoff
                    log.debug("System under stress (CPU: {}%, Queue: {}), using
longer backoff",
                        cpuUsage * 100, queueSize);
                    return new ExponentialBackOff(5000L, 2.0); // Start with 5s

                } else if (cpuUsage < 0.3 && queueSize < 100) {
                    // System has capacity - use shorter backoff
                    log.debug("System has capacity (CPU: {}%, Queue: {}), using
shorter backoff",
                        cpuUsage * 100, queueSize);
                    return new FixedBackOff(1000L, 5L); // 1s fixed

                } else {
                    // Normal load - standard exponential backoff
                    return new ExponentialBackOff(2000L, 2.0);
                }
            };

        errorHandler.setBackOffFunction(adaptiveFunction);

        configureExceptionClassification(errorHandler);
```

```
        errorHandler.setRetryListeners(createRetryListener("AdaptiveBackoff"));

        log.info("Configured AdaptiveBackoff error handler based on system
metrics");

        return errorHandler;
    }

    // Helper methods for configuration
    private void configureExceptionClassification(DefaultErrorHandler
errorHandler) {
        // Fatal exceptions - no retry
        errorHandler.addNotRetryableExceptions(
            IllegalArgumentException.class,
            NullPointerException.class,

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

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

org.springframework.messaging.handler.invocation.MethodArgumentResolutionException
.class,
            NoSuchMethodException.class,
            ClassCastException.class
        );

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

        // Reset retry state when exception type changes
        errorHandler.setResetStateOnExceptionChange(true);

        log.debug("Configured exception classification for error handler");
    }

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

                // Update metrics
```

```
                updateRetryMetrics(handlerName, record.topic(), deliveryAttempt);
            }

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

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

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

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

    // System metrics methods (would integrate with actual monitoring)
    private double getCurrentCpuUsage() {
        // Integrate with system metrics (Micrometer, JMX, etc.)
        return Math.random(); // Placeholder
    }

    private int getCurrentQueueSize() {
        // Get current processing queue size
        return (int) (Math.random() * 2000); // Placeholder
    }

    // Metrics update methods
    private void updateRetryMetrics(String handlerName, String topic, int attempt)
{
        // Update Micrometer metrics
        meterRegistry.counter("kafka.error.retry",
            Tags.of("handler", handlerName, "topic", topic, "attempt",
String.valueOf(attempt)))
            .increment();
    }

    private void updateRecoveryMetrics(String handlerName, String topic) {
        meterRegistry.counter("kafka.error.recovered",
            Tags.of("handler", handlerName, "topic", topic))
            .increment();
```

```
    }

    private void updateFailureMetrics(String handlerName, String topic) {
        meterRegistry.counter("kafka.error.failed",
            Tags.of("handler", handlerName, "topic", topic))
            .increment();
    }

    @Autowired
    private MeterRegistry meterRegistry;
}

/**
 * Container factory configurations with different error handlers
 */
@Configuration
public class ErrorHandlerContainerConfiguration {

    /**
     * Container factory with fixed backoff error handler
     */
    @Bean("fixedBackoffContainerFactory")
    public ConcurrentKafkaListenerContainerFactory<String, Object>
fixedBackoffContainerFactory(
            @Qualifier("fixedBackoffErrorHandler") CommonErrorHandler
errorHandler) {

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

        factory.setConsumerFactory(consumerFactory());
        factory.setCommonErrorHandler(errorHandler);
        factory.setConcurrency(3);

        // Manual acknowledgment for precise control

factory.getContainerProperties().setAckMode(ContainerProperties.AckMode.MANUAL_IMM
EDIATE);

        return factory;
    }

    /**
     * Container factory with exponential backoff error handler
     */
    @Bean("exponentialBackoffContainerFactory")
    public ConcurrentKafkaListenerContainerFactory<String, Object>
exponentialBackoffContainerFactory(
            @Qualifier("exponentialBackoffErrorHandler") CommonErrorHandler
errorHandler) {

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

```
        factory.setConsumerFactory(consumerFactory());
        factory.setCommonErrorHandler(errorHandler);
        factory.setConcurrency(4);


factory.getContainerProperties().setAckMode(ContainerProperties.AckMode.MANUAL_IMM
EDIATE);

        return factory;
    }

    /**
     * Container factory with custom backoff error handler
     */
    @Bean("customBackoffContainerFactory")
    public ConcurrentKafkaListenerContainerFactory<String, Object>
customBackoffContainerFactory(
            @Qualifier("customBackoffErrorHandler") CommonErrorHandler
errorHandler) {

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

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


factory.getContainerProperties().setAckMode(ContainerProperties.AckMode.MANUAL_IMM
EDIATE);

        return factory;
    }

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

/**
 * Backoff policy examples and demonstrations
 */
@Component
@lombok.extern.slf4j.Slf4j
public class BackoffPolicyExamples {

    /**
     * Fixed backoff listener - predictable retry intervals
     */
    @KafkaListener(
        topics = "fixed-backoff-topic",
        groupId = "fixed-backoff-group",
        containerFactory = "fixedBackoffContainerFactory"
```

```
    )
    public void handleFixedBackoffMessages(@Payload OrderEvent order,
                                           Acknowledgment ack) {

        log.info("Processing order with fixed backoff: orderId={}, amount={}",
            order.getOrderId(), order.getAmount());

        try {
            // Simulate business logic that might fail
            processOrderWithPossibleFailure(order);

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

        } catch (ValidationException e) {
            log.error("Validation failed for order: {} - {}", order.getOrderId(),
e.getMessage());
            throw e; // Will be retried with fixed 2s intervals

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

    /**
     * Exponential backoff listener - increasing retry intervals
     */
    @KafkaListener(
        topics = "exponential-backoff-topic",
        groupId = "exponential-backoff-group",
        containerFactory = "exponentialBackoffContainerFactory"
    )
    public void handleExponentialBackoffMessages(@Payload PaymentEvent payment,
                                                 Acknowledgment ack) {

        log.info("Processing payment with exponential backoff: paymentId={},
amount={}",
            payment.getPaymentId(), payment.getAmount());

        try {
            // Simulate external service call that might be temporarily down
            processPaymentWithExternalService(payment);

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

        } catch (ExternalServiceException e) {
            log.error("External service error for payment: {} - {}",
                payment.getPaymentId(), e.getMessage());
            throw e; // Will be retried with exponential backoff: 1s, 2s, 4s, 8s,
16s, 30s...
```

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

    /**
     * Custom backoff listener - different policies per exception type
     */
    @KafkaListener(
        topics = "custom-backoff-topic",
        groupId = "custom-backoff-group",
        containerFactory = "customBackoffContainerFactory"
    )
    public void handleCustomBackoffMessages(@Payload NotificationEvent
notification,
                                            Acknowledgment ack) {

        log.info("Processing notification with custom backoff: userId={}, type=
{}",
            notification.getUserId(), notification.getType());

        try {
            processNotificationWithMultipleServices(notification);

            ack.acknowledge();
            log.info("Successfully processed notification: userId={}",
notification.getUserId());

        } catch (ValidationException e) {
            log.error("Validation error (fast retry): userId={} - {}",
                notification.getUserId(), e.getMessage());
            throw e; // Will use FixedBackOff(500ms, 3 attempts)

        } catch (ExternalServiceException e) {
            log.error("External service error (exponential backoff): userId={} -
{}",
                notification.getUserId(), e.getMessage());
            throw e; // Will use ExponentialBackOff(2s initial, 1.5x multiplier,
60s max)

        } catch (DatabaseException e) {
            log.error("Database error (long fixed retry): userId={} - {}",
                notification.getUserId(), e.getMessage());
            throw e; // Will use FixedBackOff(5s, 10 attempts)

        } catch (Exception e) {
            log.error("Unknown error (default exponential): userId={} - {}",
                notification.getUserId(), e.getMessage());
            throw e; // Will use default ExponentialBackOff(1s, 2.0x)
        }
    }
```

```
    // Business logic simulation methods
    private void processOrderWithPossibleFailure(OrderEvent order) throws
ValidationException {
        // Simulate validation that fails 30% of the time initially
        if (order.getOrderId().hashCode() % 10 < 3) {
            throw new ValidationException("Order validation failed: " +
order.getOrderId());
        }

        // Simulate processing
        log.debug("Processing order business logic: {}", order.getOrderId());
    }

    private void processPaymentWithExternalService(PaymentEvent payment) throws
ExternalServiceException {
        // Simulate external service that's down 40% of the time
        if (payment.getPaymentId().hashCode() % 10 < 4) {
            throw new ExternalServiceException("Payment service temporarily
unavailable");
        }

        log.debug("Processing payment with external service: {}",
payment.getPaymentId());
    }

    private void processNotificationWithMultipleServices(NotificationEvent
notification)
            throws ValidationException, ExternalServiceException,
DatabaseException {

        String userId = notification.getUserId();
        int hash = userId.hashCode() % 100;

        if (hash < 10) {
            throw new ValidationException("Invalid notification format");
        } else if (hash < 25) {
            throw new ExternalServiceException("Notification service
unavailable");
        } else if (hash < 35) {
            throw new DatabaseException("Database connection failed");
        }

        log.debug("Processing notification successfully: userId={}", userId);
    }
}

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

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

```
}

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

// Domain objects
@lombok.Data
@lombok.Builder
@lombok.NoArgsConstructor
@lombok.AllArgsConstructor
class OrderEvent {
    private String orderId;
    private String customerId;
    private java.math.BigDecimal amount;
    private String status;
    private java.time.Instant timestamp;
}

@lombok.Data
@lombok.Builder
@lombok.NoArgsConstructor
@lombok.AllArgsConstructor
class PaymentEvent {
    private String paymentId;
    private String orderId;
    private java.math.BigDecimal amount;
    private String method;
    private String status;
    private java.time.Instant timestamp;
}

@lombok.Data
@lombok.Builder
@lombok.NoArgsConstructor
@lombok.AllArgsConstructor
class NotificationEvent {
    private String userId;
    private String type;
    private String message;
    private String channel;
    private java.time.Instant timestamp;
}
```

Recovery Strategies

Advanced Recovery Configuration and Patterns

```
import org.springframework.kafka.listener.ConsumerRecordRecoverer;
import org.springframework.kafka.listener.DeadLetterPublishingRecoverer;
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;

/**
 * Advanced recovery strategies for failed message handling
 */
@Configuration
@lombok.extern.slf4j.Slf4j
public class RecoveryStrategiesConfiguration {

    @Autowired
    private KafkaTemplate<String, Object> kafkaTemplate;

    @Autowired
    private MeterRegistry meterRegistry;

    /**
     * Simple logging recoverer - just logs failed messages
     */
    @Bean("loggingRecoverer")
    public ConsumerRecordRecoverer loggingRecoverer() {
        return (record, exception) -> {
            log.error("Failed to process record after all retries: topic={},
partition={}, offset={}, key={}, error={}",
                record.topic(), record.partition(), record.offset(), record.key(),
exception.getMessage());

            // Update failure metrics
            meterRegistry.counter("kafka.message.failed",
                Tags.of("topic", record.topic(), "reason", "exhausted_retries"))
                .increment();
        };
    }

    /**
     * Dead letter topic recoverer with custom destination resolution
     */
    @Bean("deadLetterRecoverer")
    public DeadLetterPublishingRecoverer deadLetterRecoverer() {

        // Custom destination resolver based on exception type and record content
        BiFunction<ConsumerRecord<?, ?>, Exception, TopicPartition>
destinationResolver =
            (record, exception) -> {

                String originalTopic = record.topic();
                int originalPartition = record.partition();

                // Route to different DLT topics based on exception type
                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 {
                    // Default DLT for unknown errors
                    return new TopicPartition(originalTopic + ".DLT",
originalPartition);
                }
            };

        DeadLetterPublishingRecoverer recoverer = new
DeadLetterPublishingRecoverer(
            kafkaTemplate, destinationResolver);

        // Add custom headers to DLT messages
        recoverer.setHeadersFunction((record, ex) -> {
            Map<String, Object> headers = new HashMap<>();
            headers.put("original-topic", record.topic());
            headers.put("original-partition", record.partition());
            headers.put("original-offset", record.offset());
            headers.put("failure-timestamp", System.currentTimeMillis());
            headers.put("exception-class", ex.getClass().getName());
            headers.put("exception-message", ex.getMessage());
            headers.put("retry-count", getRetryCount(record));

            // Add business context headers if available
            if (record.value() instanceof OrderEvent order) {
                headers.put("order-id", order.getOrderId());
                headers.put("customer-id", order.getCustomerId());
            } else if (record.value() instanceof PaymentEvent payment) {
                headers.put("payment-id", payment.getPaymentId());
                headers.put("payment-method", payment.getMethod());
            }

            return headers;
        });

        log.info("Configured DeadLetterPublishingRecoverer with custom destination
resolution");

        return recoverer;
    }
```

```
    /**
     * Composite recoverer - tries multiple recovery strategies
     */
    @Bean("compositeRecoverer")
    public ConsumerRecordRecoverer compositeRecoverer() {
        return (record, exception) -> {
            log.info("Starting composite recovery for record: topic={}, partition=
{}, offset={}",
                record.topic(), record.partition(), record.offset());

            try {
                // Strategy 1: Try to recover based on message content
                if (attemptBusinessLogicRecovery(record, exception)) {
                    log.info("Business logic recovery successful: topic={},
offset={}",
                        record.topic(), record.offset());
                    return;
                }

                // Strategy 2: Try to send to retry topic (with delay)
                if (attemptRetryTopicRecovery(record, exception)) {
                    log.info("Retry topic recovery successful: topic={}, offset=
{}",
                        record.topic(), record.offset());
                    return;
                }

                // Strategy 3: Send to appropriate DLT
                attemptDeadLetterRecovery(record, exception);

            } catch (Exception recoveryException) {
                log.error("All recovery strategies failed: topic={}, offset={}",
                    record.topic(), record.offset(), recoveryException);

                // Final fallback - just log
                loggingRecoverer().accept(record, exception);
            }
        };
    }

    /**
     * Selective recoverer - different strategies based on message type
     */
    @Bean("selectiveRecoverer")
    public ConsumerRecordRecoverer selectiveRecoverer() {
        return (record, exception) -> {
            Object value = record.value();

            log.info("Selective recovery for message type: {}",
value.getClass().getSimpleName());

            if (value instanceof OrderEvent order) {
                recoverOrderEvent(record, order, exception);
```

```
            } else if (value instanceof PaymentEvent payment) {
                recoverPaymentEvent(record, payment, exception);

            } else if (value instanceof NotificationEvent notification) {
                recoverNotificationEvent(record, notification, exception);

            } else {
                // Unknown message type - use default recovery
                deadLetterRecoverer().accept(record, exception);
            }
        };
    }

    /**
     * Conditional recoverer - recover only certain types of failures
     */
    @Bean("conditionalRecoverer")
    public ConsumerRecordRecoverer conditionalRecoverer() {
        return (record, exception) -> {

            // Check if this is a recoverable failure
            if (isRecoverableFailure(exception)) {
                log.info("Attempting recovery for recoverable failure: topic={},
error={}",
                    record.topic(), exception.getMessage());

                // Send to retry topic with delay headers
                sendToRetryTopic(record, exception);

            } else if (isPoisonPill(exception)) {
                log.warn("Poison pill detected, sending to quarantine: topic={},
offset={}",
                    record.topic(), record.offset());

                // Send to quarantine topic for manual inspection
                sendToQuarantineTopic(record, exception);

            } else {
                log.error("Non-recoverable failure, sending to DLT: topic={},
error={}",
                    record.topic(), exception.getMessage());

                // Send to DLT
                deadLetterRecoverer().accept(record, exception);
            }
        };
    }

    /**
     * Metrics-enhanced recoverer - tracks recovery patterns
     */
    @Bean("metricsRecoverer")
    public ConsumerRecordRecoverer metricsRecoverer() {
```

```
        return (record, exception) -> {
            String topic = record.topic();
            String exceptionType = exception.getClass().getSimpleName();

            // Track recovery metrics
            meterRegistry.counter("kafka.recovery.attempts",
                Tags.of("topic", topic, "exception", exceptionType))
                .increment();

            Timer.Sample sample = Timer.start(meterRegistry);

            try {
                // Perform recovery
                deadLetterRecoverer().accept(record, exception);

                // Track successful recovery
                meterRegistry.counter("kafka.recovery.success",
                    Tags.of("topic", topic, "exception", exceptionType))
                    .increment();

            } catch (Exception recoveryException) {
                // Track recovery failures
                meterRegistry.counter("kafka.recovery.failed",
                    Tags.of("topic", topic, "exception", exceptionType))
                    .increment();

                throw recoveryException;

            } finally {
                sample.stop(Timer.builder("kafka.recovery.duration")
                    .tag("topic", topic)
                    .tag("exception", exceptionType)
                    .register(meterRegistry));
            }
        };
    }

    // Helper methods for recovery strategies
    private boolean attemptBusinessLogicRecovery(ConsumerRecord<?, ?> record,
Exception exception) {
        try {
            Object value = record.value();

            // Try to fix the data and reprocess
            if (value instanceof OrderEvent order && exception instanceof
ValidationException) {
                // Try to fix validation issues
                OrderEvent fixedOrder = fixOrderValidation(order);
                if (fixedOrder != null) {
                    // Send fixed order back to original topic
                    kafkaTemplate.send(record.topic(), record.key(), fixedOrder);
                    log.info("Fixed and resubmitted order: {}",
order.getOrderId());
                    return true;
```

```
                }
            }

            return false;

        } catch (Exception e) {
            log.debug("Business logic recovery failed", e);
            return false;
        }
    }

    private boolean attemptRetryTopicRecovery(ConsumerRecord<?, ?> record,
Exception exception) {
        try {
            // Only retry certain types of exceptions
            if (exception instanceof ExternalServiceException ||
                exception instanceof java.util.concurrent.TimeoutException) {

                String retryTopic = record.topic() + ".retry";

                // Add delay headers for delayed processing
                ProducerRecord<Object, Object> retryRecord = new ProducerRecord<>(
                    retryTopic, record.partition(), record.key(), record.value());

                // Add retry metadata
                retryRecord.headers().add("original-topic",
record.topic().getBytes());
                retryRecord.headers().add("retry-timestamp",
                    String.valueOf(System.currentTimeMillis() +
60000).getBytes()); // 1 minute delay
                retryRecord.headers().add("retry-reason",
exception.getMessage().getBytes());

                kafkaTemplate.send(retryRecord);

                log.info("Sent to retry topic: original={}, retry={}",
record.topic(), retryTopic);
                return true;
            }

            return false;

        } catch (Exception e) {
            log.debug("Retry topic recovery failed", e);
            return false;
        }
    }

    private void attemptDeadLetterRecovery(ConsumerRecord<?, ?> record, Exception
exception) {
        deadLetterRecoverer().accept(record, exception);
        log.info("Sent to dead letter topic: topic={}, offset={}", record.topic(),
record.offset());
    }
```

```
    private void recoverOrderEvent(ConsumerRecord<?, ?> record, OrderEvent order,
Exception exception) {
        log.info("Recovering order event: orderId={}, error={}",
order.getOrderId(), exception.getMessage());

        if (exception instanceof ValidationException) {
            // Try to fix validation and resubmit
            OrderEvent fixedOrder = fixOrderValidation(order);
            if (fixedOrder != null) {
                kafkaTemplate.send(record.topic(), record.key(), fixedOrder);
                return;
            }
        }

        // Send to order-specific DLT
        String dltTopic = record.topic() + ".orders.DLT";
        kafkaTemplate.send(dltTopic, record.key(), order);
    }

    private void recoverPaymentEvent(ConsumerRecord<?, ?> record, PaymentEvent
payment, Exception exception) {
        log.info("Recovering payment event: paymentId={}, error={}",
payment.getPaymentId(), exception.getMessage());

        // Payments are critical - send to high-priority DLT
        String dltTopic = record.topic() + ".payments.priority.DLT";

        ProducerRecord<Object, Object> dltRecord = new ProducerRecord<>(dltTopic,
record.key(), payment);
        dltRecord.headers().add("priority", "HIGH".getBytes());
        dltRecord.headers().add("payment-method", payment.getMethod().getBytes());

        kafkaTemplate.send(dltRecord);
    }

    private void recoverNotificationEvent(ConsumerRecord<?, ?> record,
NotificationEvent notification, Exception exception) {
        log.info("Recovering notification event: userId={}, type={}, error={}",
            notification.getUserId(), notification.getType(),
exception.getMessage());

        // Notifications are less critical - can be dropped or sent to low-
priority DLT
        if (notification.getType().equals("MARKETING")) {
            log.info("Dropping marketing notification after failure: userId={}",
notification.getUserId());
            return; // Just drop marketing notifications
        }

        // Send important notifications to DLT
        String dltTopic = record.topic() + ".notifications.DLT";
        kafkaTemplate.send(dltTopic, record.key(), notification);
    }
```

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

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

    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 metadata
        retryRecord.headers().add("retry-delay-until",
            String.valueOf(System.currentTimeMillis() + 300000).getBytes()); // 5
minutes
        retryRecord.headers().add("original-failure",
exception.getMessage().getBytes());

        kafkaTemplate.send(retryRecord);
    }

    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 metadata
        quarantineRecord.headers().add("quarantine-reason",
"POISON_PILL".getBytes());
        quarantineRecord.headers().add("original-exception",
exception.getClass().getName().getBytes());
        quarantineRecord.headers().add("quarantine-timestamp",
            String.valueOf(System.currentTimeMillis()).getBytes());

        kafkaTemplate.send(quarantineRecord);
    }

    private OrderEvent fixOrderValidation(OrderEvent order) {
        try {
            // Attempt to fix common validation issues
```

```
            OrderEvent.OrderEventBuilder builder = order.toBuilder();

            // Fix null or empty fields
            if (order.getOrderId() == null || order.getOrderId().isEmpty()) {
                builder.orderId("FIXED_" + UUID.randomUUID().toString());
            }

            if (order.getAmount() == null ||
order.getAmount().compareTo(java.math.BigDecimal.ZERO) <= 0) {
                return null; // Can't fix invalid amount
            }

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

            if (order.getTimestamp() == null) {
                builder.timestamp(java.time.Instant.now());
            }

            OrderEvent fixedOrder = builder.build();
            log.info("Fixed order validation: original={}, fixed={}",
order.getOrderId(), fixedOrder.getOrderId());

            return fixedOrder;

        } catch (Exception e) {
            log.debug("Could not fix order validation", e);
            return null;
        }
    }

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

This completes the first major section of the Spring Kafka Error Handling guide, covering DefaultErrorHandler with comprehensive backoff policies and recovery strategies. The guide provides extensive Java examples and production-ready patterns for handling various failure scenarios.