Spring Kafka Error Handling & Retry Cheat Sheet - Master Level

5.1 DefaultErrorHandler (Spring Kafka 2.8+)

Definition DefaultErrorHandler provides comprehensive error handling capabilities in Spring Kafka 2.8+ with configurable retry mechanisms, backoff policies, and recovery strategies while integrating with Spring's exception handling infrastructure and container lifecycle management. The handler coordinates retry attempts, exception classification, and recovery procedures while maintaining consumer session health and partition assignment coordination for reliable error processing and operational continuity.

Key Highlights Configurable retry attempts with sophisticated backoff policies including fixed delay, exponential backoff, and custom timing strategies while supporting exception classification for different error handling approaches. Integration with Dead Letter Topic publishing enables unprocessable message isolation while maintaining error context and metadata for operational analysis and manual intervention procedures. Container lifecycle coordination ensures error handling doesn't interfere with consumer group membership while providing comprehensive logging and monitoring capabilities for production error tracking and analysis.

Responsibility / Role Error handling coordination manages exception processing and retry attempts while maintaining consumer session health and partition assignment semantics for reliable error recovery without affecting consumer group stability. Retry coordination implements backoff policies and attempt counting while providing exception classification and routing for different error types and recovery strategies based on business requirements. Recovery strategy execution handles final error disposition including Dead Letter Topic publishing, logging, and custom recovery logic while maintaining error context and operational metadata for troubleshooting and analysis.

Underlying Data Structures / Mechanism Error handler implementation uses retry state management with attempt counting and backoff timing while coordinating with container message processing and consumer offset management for consistent error handling behavior. Exception classification uses configurable exception hierarchies while retry state persistence enables sophisticated retry patterns and error tracking across processing attempts and container restarts. Integration with Spring's scheduling infrastructure provides accurate backoff timing while maintaining thread safety and concurrent error processing capabilities for high-throughput scenarios.

Advantages Comprehensive error handling capabilities eliminate need for custom error processing logic while providing production-grade retry mechanisms and recovery strategies for reliable message processing patterns. Spring integration provides consistent configuration and lifecycle management while maintaining compatibility with transaction boundaries and container coordination for enterprise deployment scenarios. Flexible retry policies enable optimization for different error types while comprehensive logging and monitoring provide operational visibility and troubleshooting capabilities for production error management.

Disadvantages / Trade-offs Error handling overhead can affect consumer performance while retry attempts increase processing latency and resource utilization requiring careful configuration and monitoring for optimal characteristics. Complex error scenarios require sophisticated classification and recovery strategies while debugging error handling issues can be challenging due to retry state management and timing

complexity. Resource consumption increases with retry state maintenance while concurrent error processing can cause thread pool pressure requiring capacity planning and resource allocation optimization.

Corner Cases Container shutdown during retry processing can cause incomplete error handling while consumer rebalancing during retry attempts can cause partition assignment conflicts and retry state loss requiring coordination procedures. Exception classification conflicts can cause unexpected retry behavior while backoff timing coordination with consumer session timeout can cause rebalancing issues requiring careful timeout configuration and monitoring. Error handler failures can cause processing deadlock while retry state corruption can cause inconsistent error handling behavior requiring recovery procedures and state validation mechanisms.

Limits / Boundaries Maximum retry attempts typically configured between 3-10 attempts while backoff duration ranges from milliseconds to minutes depending on error type and business requirements affecting processing latency and resource utilization. Retry state memory usage scales with concurrent error processing while error handler thread pool capacity affects concurrent error handling performance and resource allocation. Container integration limits error handling scope to partition boundaries while consumer session timeout constraints affect maximum retry duration and error processing timing coordination.

Default Values DefaultErrorHandler uses 10 retry attempts with exponential backoff starting at 1 second while exception classification treats all exceptions as retriable requiring explicit configuration for production error handling strategies. Backoff multiplier defaults to 2.0 with maximum backoff of 30 seconds while Dead Letter Topic publishing is disabled by default requiring explicit configuration for error isolation and recovery procedures.

Best Practices Configure retry attempts and backoff policies based on error characteristics and business requirements while implementing appropriate exception classification for different error types and recovery strategies. Monitor error handling performance and retry success rates while implementing comprehensive logging and alerting for error patterns and operational analysis requiring attention. Design error handling strategies with consumer session timeout in mind while coordinating retry timing with container lifecycle and partition assignment stability ensuring reliable error processing and consumer group health.

5.1.1 Backoff policies (fixed, exponential)

Definition Backoff policies in DefaultErrorHandler control retry timing through fixed delay or exponential backoff strategies with configurable parameters including initial delay, multiplier, and maximum backoff duration for optimal error handling performance. Policy selection affects retry timing characteristics while providing protection against overwhelming downstream systems during error scenarios and enabling sophisticated retry strategies based on error types and business requirements.

Key Highlights Fixed delay backoff provides consistent retry timing while exponential backoff implements escalating delays with configurable multiplier and maximum duration for protecting downstream systems from retry storms. Configurable jitter addition prevents thundering herd effects while backoff policy selection can be customized per exception type for sophisticated error handling strategies. Integration with retry attempt limits while backoff timing coordination ensures retry attempts complete within consumer session timeout boundaries maintaining consumer group stability.

Responsibility / Role Backoff timing coordination manages delay calculation and execution while ensuring retry attempts don't exceed consumer session timeout limits and cause partition assignment issues. Policy implementation provides consistent timing behavior while supporting different strategies for various error

types and downstream system protection requirements. Integration with retry state management ensures accurate timing while coordinating with container lifecycle and consumer session health for reliable error processing patterns.

Underlying Data Structures / Mechanism Backoff calculation uses configurable algorithms with timing state management while exponential backoff implements geometric progression with optional jitter and maximum duration limits. Timing execution uses Spring's scheduling infrastructure while maintaining thread safety and concurrent backoff processing capabilities for multiple error scenarios. State management tracks retry attempts and timing while coordinating with container message processing and consumer session management for optimal error handling coordination.

Advantages Fixed delay provides predictable retry timing while exponential backoff protects downstream systems through escalating delays and automatic protection against retry storms during systematic failures. Configurable parameters enable optimization for different error patterns while jitter addition prevents synchronized retry attempts across multiple consumer instances. Integration with consumer session management ensures backoff timing doesn't cause partition assignment issues while maintaining reliable error processing and recovery capabilities.

Disadvantages / Trade-offs Fixed delay may not be optimal for all error types while exponential backoff can cause extended processing delays for transient failures requiring careful balance between protection and responsiveness. Backoff timing increases error processing latency while longer delays can cause consumer session timeout if not properly coordinated with container configuration. Complex backoff strategies require careful tuning while backoff state management increases memory usage and processing overhead affecting error handling performance.

Corner Cases Exponential backoff reaching maximum duration can cause extended processing delays while backoff timing coordination with container shutdown can cause incomplete retry processing and resource cleanup issues. Consumer rebalancing during backoff delays can cause retry state loss while backoff jitter calculation can cause timing precision issues requiring careful configuration and testing. System clock changes can affect backoff timing while thread interruption during backoff delays can cause retry state inconsistency requiring comprehensive error handling and recovery procedures.

Limits / Boundaries Backoff duration typically ranges from 100ms to several minutes while exponential backoff multiplier usually configured between 1.5-3.0 for optimal protection without excessive delays. Maximum backoff duration often limited to consumer session timeout fraction while jitter percentage typically ranges from 10-50% for effective thundering herd prevention. Backoff precision limited by system timer resolution while concurrent backoff processing constrained by available thread pool capacity and scheduling infrastructure performance.

Default Values Fixed delay backoff uses 1 second default delay while exponential backoff starts at 1 second with 2.0 multiplier and 30 second maximum duration. Jitter is disabled by default while backoff policy applies to all exceptions requiring explicit configuration for exception-specific backoff strategies and production optimization requirements.

Best Practices Configure backoff policies based on downstream system characteristics and error patterns while implementing exponential backoff for protecting external systems from retry storms during systematic failures. Monitor backoff effectiveness and retry success rates while tuning parameters for optimal balance between protection and responsiveness based on error characteristics and business requirements. Coordinate backoff timing with consumer session timeout while implementing appropriate jitter for preventing

synchronized retry attempts across distributed consumer instances ensuring reliable error processing and system protection.

5.1.2 Recovery strategies

Definition Recovery strategies in DefaultErrorHandler define final error disposition after retry exhaustion including Dead Letter Topic publishing, custom recovery logic, and error logging with comprehensive error context preservation for operational analysis. Strategy configuration enables different recovery approaches based on exception types while maintaining error metadata and enabling sophisticated error handling patterns for production deployment and operational requirements.

Key Highlights Dead Letter Topic publishing provides automatic error isolation while custom recovery strategies enable application-specific error handling and business logic integration for sophisticated error processing patterns. Error context preservation maintains original message metadata, exception details, and retry attempt history while recovery strategy selection can be customized per exception type and business requirements. Integration with Spring's exception handling infrastructure while recovery execution maintains transaction boundaries and container lifecycle coordination for reliable error disposition and operational continuity.

Responsibility / Role Recovery strategy coordination manages final error disposition while maintaining error context and metadata for operational analysis and potential manual intervention procedures. Error isolation through Dead Letter Topic publishing while custom recovery logic enables business-specific error handling and integration with external systems for comprehensive error management. Logging and monitoring coordination provides operational visibility while recovery strategy execution maintains container health and consumer session stability during error processing scenarios.

Underlying Data Structures / Mechanism Recovery strategy implementation uses configurable strategy interfaces while error context preservation maintains message metadata, exception hierarchy, and retry attempt details for comprehensive error analysis. Dead Letter Topic publishing uses specialized producers while maintaining error context serialization and topic configuration for reliable error isolation and operational procedures. Custom recovery integration uses Spring's callback mechanisms while maintaining transaction coordination and container lifecycle management for consistent error processing behavior.

Advantages Flexible recovery options enable optimization for different business requirements while Dead Letter Topic integration provides reliable error isolation and operational analysis capabilities for production error management. Custom recovery strategies support sophisticated business logic while error context preservation enables comprehensive troubleshooting and operational monitoring for error pattern analysis. Spring integration provides consistent configuration and lifecycle management while maintaining transaction boundaries and container coordination for enterprise deployment scenarios.

Disadvantages / Trade-offs Recovery strategy complexity can affect error processing performance while Dead Letter Topic publishing increases infrastructure requirements and operational complexity for comprehensive error management. Custom recovery logic requires additional development and testing while recovery strategy failures can cause error handling deadlock requiring comprehensive error handling and recovery procedures. Resource utilization increases with error context preservation while complex recovery strategies can cause processing bottlenecks requiring optimization and monitoring.

Corner Cases Dead Letter Topic unavailability can cause recovery failures while custom recovery strategy exceptions can cause error handling recursion requiring comprehensive error handling and circuit breaker

patterns. Recovery strategy execution during container shutdown can cause incomplete error processing while transaction rollback during recovery can cause inconsistent error disposition requiring coordination procedures. Error context serialization failures while recovery strategy configuration conflicts can cause unexpected error handling behavior requiring validation and testing procedures.

Limits / Boundaries Recovery strategy execution time constrained by container lifecycle while custom recovery complexity bounded by available system resources and integration capabilities with external systems. Dead Letter Topic capacity and retention policies while error context size affects serialization performance and storage requirements for comprehensive error preservation. Maximum recovery strategy count per error handler while strategy selection overhead scales with exception classification complexity affecting error processing performance.

Default Values Recovery strategy defaults to logging with error context while Dead Letter Topic publishing requires explicit configuration including topic naming and producer setup for error isolation procedures. Custom recovery strategies require explicit implementation while error context preservation follows default serialization patterns requiring customization for production error analysis and operational requirements.

Best Practices Design recovery strategies based on business requirements and operational capabilities while implementing Dead Letter Topic publishing for unprocessable messages requiring manual intervention and analysis procedures. Configure comprehensive error context preservation while monitoring recovery strategy effectiveness and error patterns for operational analysis and system improvement identification. Implement appropriate error isolation and recovery procedures while coordinating with operational monitoring and alerting systems ensuring effective error management and business continuity across error scenarios and system failures.

5.2 Dead Letter Topics (DLT)

5.2.1 Configuring DLT publishing

Definition Dead Letter Topic publishing configuration enables automatic routing of unprocessable messages to dedicated error topics with comprehensive error context preservation including original message metadata, exception details, and processing history for operational analysis. DLT setup integrates with error handling strategies while providing configurable topic naming, producer configuration, and error context serialization for reliable error isolation and operational procedures.

Key Highlights Automatic topic creation with configurable naming patterns while error context enrichment includes original message headers, exception stack traces, and retry attempt history for comprehensive troubleshooting and analysis capabilities. Producer configuration provides independent settings for DLT publishing while maintaining transaction coordination and exactly-once semantics for reliable error message delivery and operational consistency. Integration with Spring Boot auto-configuration while supporting custom serializers and topic configuration for enterprise deployment and error management requirements.

Responsibility / Role DLT publishing coordination manages error message routing while maintaining comprehensive error context and metadata preservation for operational analysis and potential message recovery procedures. Topic management handles automatic creation and configuration while producer coordination ensures reliable error message delivery with appropriate serialization and durability guarantees. Error context serialization manages complex error metadata while maintaining compatibility with operational tooling and error analysis procedures for comprehensive error management.

Underlying Data Structures / Mechanism DLT publisher implementation uses dedicated Kafka producers while error context serialization maintains message metadata, exception details, and processing history through configurable serialization strategies. Topic naming uses pattern-based generation while producer configuration provides independent settings for error message delivery and durability characteristics. Error context enrichment uses structured metadata formats while maintaining compatibility with operational analysis tools and error recovery procedures for comprehensive error management.

Advantages Reliable error isolation prevents unprocessable messages from affecting normal processing while comprehensive error context enables effective troubleshooting and operational analysis for production error management. Automatic topic management eliminates manual DLT setup while configurable naming patterns support organizational standards and operational procedures for error topic organization. Independent producer configuration enables optimization for error handling characteristics while maintaining transaction coordination and exactly-once semantics for reliable error message delivery.

Disadvantages / Trade-offs Additional infrastructure requirements for DLT topics while producer overhead increases resource utilization and operational complexity for comprehensive error handling and isolation procedures. Error context serialization can cause significant overhead while DLT message size may be substantially larger than original messages affecting storage and network utilization. Topic proliferation with automatic creation while DLT producer failures can cause error handling failures requiring comprehensive error handling and recovery procedures for operational reliability.

Corner Cases DLT topic unavailability can cause error handling failures while topic creation failures can prevent error isolation requiring comprehensive error handling and recovery procedures for operational continuity. Error context serialization failures can cause DLT publishing errors while producer configuration conflicts can affect error message delivery requiring validation and testing procedures. Topic naming conflicts while DLT producer authentication failures can cause error handling degradation requiring operational coordination and monitoring procedures.

Limits / Boundaries DLT message size limited by Kafka message size limits while error context serialization affects message size and processing performance requiring optimization and potentially context reduction strategies. Producer configuration complexity while DLT topic retention policies affect error message availability for operational analysis and recovery procedures requiring coordination with organizational retention requirements. Maximum error context size while serialization performance scales with context complexity affecting error handling throughput and resource utilization characteristics.

Default Values DLT publishing requires explicit configuration while topic naming follows pattern-based defaults using original topic name with suffix conventions for organizational standards. Producer configuration inherits container defaults while error context serialization uses basic metadata preservation requiring customization for comprehensive error analysis and operational requirements.

Best Practices Configure DLT topics with appropriate retention policies while implementing comprehensive error context preservation enabling effective troubleshooting and operational analysis for production error management procedures. Design topic naming strategies based on organizational standards while monitoring DLT publishing performance and error message characteristics for operational optimization and resource planning. Implement appropriate access controls and monitoring for DLT topics while coordinating with operational procedures for error analysis and potential message recovery ensuring effective error management and business continuity.

5.2.2 Retrying from DLT

Definition DLT retry processing enables reprocessing of error messages from Dead Letter Topics through dedicated consumer applications or manual intervention procedures with support for error correction, data transformation, and selective message recovery. Retry mechanisms coordinate with original message processing while maintaining error context and enabling sophisticated recovery strategies for business continuity and operational error management requirements.

Key Highlights Dedicated DLT consumer applications enable automated retry processing while manual intervention tools support selective message recovery and error correction procedures for comprehensive error management and operational flexibility. Error context preservation maintains original processing metadata while retry coordination supports message transformation and error correction before resubmission to original processing topics. Integration with monitoring and alerting systems while retry processing maintains exactly-once semantics and transaction coordination for reliable message recovery and business continuity.

Responsibility / Role DLT consumer coordination manages error message retrieval while supporting selective processing and message filtering for targeted retry operations and error correction procedures. Retry logic implements error correction strategies while maintaining original message semantics and supporting data transformation for addressing root cause issues and systematic errors. Recovery coordination manages message resubmission while maintaining processing guarantees and integration with original processing systems for seamless error recovery and business continuity.

Underlying Data Structures / Mechanism DLT consumer implementation uses standard Kafka consumer APIs while error context deserialization reconstructs original message metadata and processing history for retry decision making and error correction procedures. Message transformation uses configurable processing pipelines while retry coordination maintains transaction boundaries and exactly-once processing guarantees for reliable message recovery. Recovery state management tracks retry attempts while coordinating with operational monitoring and business continuity procedures for comprehensive error management.

Advantages Automated error recovery reduces operational overhead while selective retry processing enables targeted error correction and systematic issue resolution for comprehensive error management and business continuity. Error context availability enables sophisticated retry strategies while message transformation supports error correction and data quality improvement during recovery procedures. Integration with existing processing infrastructure while retry coordination maintains processing guarantees and transaction boundaries for reliable message recovery and operational consistency.

Disadvantages / Trade-offs Additional infrastructure requirements for DLT processing while retry logic complexity increases development and operational overhead for comprehensive error recovery and management procedures. Retry processing can cause duplicate message scenarios while error correction logic requires comprehensive testing and validation procedures for reliable recovery and data integrity assurance. Resource utilization increases with DLT processing while retry coordination complexity affects operational procedures and monitoring requirements for effective error management.

Corner Cases DLT message corruption can prevent retry processing while error context deserialization failures can cause retry errors requiring comprehensive error handling and recovery procedures for operational continuity. Retry processing failures can cause message loss while duplicate retry attempts can cause data consistency issues requiring coordination procedures and monitoring. Error correction logic failures while retry coordination with original processing can cause processing conflicts requiring comprehensive testing and validation procedures.

Limits / Boundaries DLT processing throughput limited by consumer capacity while retry logic complexity affects processing performance and resource utilization requiring optimization and capacity planning for operational requirements. Message transformation capabilities bounded by available processing resources while error correction sophistication limited by business logic complexity and operational procedures. Maximum retry attempts while DLT retention policies affect message availability for recovery procedures requiring coordination with organizational retention and business continuity requirements.

Default Values DLT retry processing requires explicit implementation while error context deserialization follows standard serialization patterns requiring customization for comprehensive error recovery and operational procedures. Retry coordination uses standard consumer configuration while recovery logic requires application-specific implementation based on business requirements and error management strategies.

Best Practices Design DLT retry processing with appropriate selectivity while implementing comprehensive error correction logic and data transformation capabilities for effective error recovery and business continuity procedures. Monitor DLT processing performance while implementing appropriate retry limits and error handling for systematic issues and operational analysis enabling continuous improvement. Coordinate DLT retry procedures with operational monitoring while implementing comprehensive testing and validation ensuring reliable error recovery and data integrity across business continuity scenarios and operational requirements.

5.3 SeekToCurrentErrorHandler (legacy)

Definition SeekToCurrentErrorHandler provides legacy error handling capabilities in older Spring Kafka versions through record seeking and retry mechanisms with configurable attempt limits and recovery strategies for backward compatibility. The handler coordinates consumer offset management with retry attempts while maintaining partition assignment and consumer group membership for legacy system integration and migration scenarios requiring error handling capabilities.

Key Highlights Consumer seek operations enable record replay while retry attempt coordination maintains processing order and consumer offset management for reliable error handling in legacy deployment scenarios. Configurable retry limits with exception classification while recovery strategy integration supports Dead Letter Topic publishing and custom recovery logic for comprehensive error management. Legacy compatibility with older Spring Kafka versions while providing migration path to newer error handling mechanisms and modern error processing strategies.

Responsibility / Role Error handling coordination manages record seeking and retry attempts while maintaining consumer session health and partition assignment for reliable error processing without affecting consumer group stability in legacy environments. Offset management coordinates seek operations with retry state while ensuring consumer position consistency and preventing message loss during error handling and recovery procedures. Recovery strategy execution handles final error disposition while maintaining error context and operational metadata for troubleshooting and migration planning.

Underlying Data Structures / Mechanism Seek operation uses consumer client offset management while retry state tracking coordinates attempt counting and error classification for consistent error handling behavior across processing attempts. Consumer coordination maintains session health while seek operations preserve partition assignment and consumer group membership during error handling and retry scenarios. Error context preservation uses configurable serialization while recovery strategy execution maintains container lifecycle coordination for reliable error processing.

Advantages Legacy system compatibility enables error handling in older Spring Kafka deployments while providing migration foundation for upgrading to modern error handling mechanisms and processing strategies. Record replay capability ensures no message loss while retry coordination maintains processing order and consumer group stability for reliable error processing in legacy environments. Configurable retry and recovery strategies provide flexibility while maintaining compatibility with existing infrastructure and operational procedures.

Disadvantages / Trade-offs Limited error handling capabilities compared to modern alternatives while seek operations can cause performance overhead and potential processing delays affecting consumer throughput and system performance. Consumer offset management complexity while legacy architecture limitations affect error handling sophistication and integration with modern operational tools and monitoring systems. Migration overhead to modern error handling while legacy system constraints limit error processing optimization and operational visibility.

Corner Cases Seek operation failures can cause processing deadlock while consumer rebalancing during error handling can cause partition assignment conflicts and error handling state loss requiring recovery procedures. Legacy container lifecycle issues while error handling during shutdown can cause incomplete processing and resource cleanup requiring coordination procedures and monitoring. Migration timing coordination while legacy error handling conflicts with modern infrastructure can cause processing issues requiring careful planning and testing procedures.

Limits / Boundaries Retry attempt limits typically configured lower than modern alternatives while seek operation performance affects consumer throughput and processing latency requiring optimization for legacy system characteristics. Error handling complexity limited by legacy architecture while integration capabilities constrained by older Spring Kafka versions affecting operational procedures and monitoring integration. Consumer coordination overhead while legacy infrastructure limitations affect error handling scalability and performance characteristics.

Default Values SeekToCurrentErrorHandler uses basic retry configuration while error classification follows legacy exception handling patterns requiring explicit configuration for production error management and operational procedures. Recovery strategies require explicit implementation while legacy defaults provide minimal error handling capabilities requiring customization for comprehensive error management.

Best Practices Plan migration to modern error handling mechanisms while maintaining SeekToCurrentErrorHandler for legacy system compatibility and gradual migration procedures ensuring business continuity and operational stability. Configure appropriate retry limits and recovery strategies while monitoring legacy error handling performance and planning modernization efforts for improved error management capabilities. Implement comprehensive error handling procedures while coordinating with migration planning ensuring reliable error processing during transition periods and system upgrade scenarios.