Spring Kafka Producer Side Cheat Sheet - Master Level

2.1 KafkaTemplate

2.1.1 Synchronous vs Asynchronous send

Definition KafkaTemplate supports both synchronous sending with immediate result blocking and asynchronous sending with callback-based result handling, enabling different performance and reliability patterns for message publishing. Synchronous operations use send().get() pattern for immediate confirmation while asynchronous operations use callback mechanisms or CompletableFuture integration for non-blocking publishing patterns with better throughput characteristics.

Key Highlights Synchronous sending blocks thread execution until broker acknowledgment providing immediate error handling and delivery confirmation but limiting throughput due to thread blocking and request serialization. Asynchronous sending enables higher throughput through non-blocking operations while requiring callback handling for error processing and delivery confirmation with potential complexity in error recovery scenarios. Performance characteristics differ significantly with async operations supporting 5-10x higher throughput while sync operations provide simpler error handling and immediate result processing.

Responsibility / Role Synchronous sending coordinates immediate result processing with thread blocking while providing simple error handling and delivery confirmation for applications requiring immediate feedback and low-latency response patterns. Asynchronous sending manages callback registration and execution while coordinating with Spring's async infrastructure and thread pools for high-throughput scenarios and non-blocking application patterns. Error handling coordination differs between patterns with sync providing immediate exception propagation while async requires callback-based error processing and recovery strategies.

Underlying Data Structures / Mechanism Synchronous implementation uses Future.get() blocking pattern while coordinating with underlying Kafka producer send operation and broker acknowledgment cycles for immediate result availability. Asynchronous implementation uses callback registration with ListenableFuture or CompletableFuture integration while managing callback execution on appropriate thread pools and error propagation mechanisms. Spring integration provides async template methods with callback coordination while maintaining compatibility with Spring's async processing infrastructure and reactive programming models.

Advantages Synchronous sending provides immediate error handling and simplified programming model while ensuring delivery confirmation before method return enabling straightforward error recovery and application logic flow. Asynchronous sending enables significantly higher throughput while supporting non-blocking application patterns and better resource utilization through concurrent processing and reduced thread blocking. Flexibility to choose appropriate pattern based on application requirements while maintaining consistent KafkaTemplate API surface and Spring integration patterns.

Disadvantages / Trade-offs Synchronous sending severely limits throughput due to thread blocking while increasing latency and resource utilization through thread waiting and request serialization affecting application scalability. Asynchronous sending increases complexity through callback handling while error recovery becomes more complex requiring sophisticated callback logic and potentially custom error handling

strategies. Memory usage increases with async operations due to callback registration while debugging becomes more complex with asynchronous execution patterns and callback coordination.

Corner Cases Synchronous timeout configuration can cause indefinite blocking while network issues can cause thread exhaustion and application deadlock requiring careful timeout management and thread pool sizing. Asynchronous callback execution failures can cause silent errors while callback thread pool exhaustion can cause callback delays and potential memory leaks requiring comprehensive error handling and resource management. Mixed sync/async usage patterns can cause unexpected behavior while Spring transaction boundaries may not coordinate properly with async publishing requiring careful transaction design.

Limits / Boundaries Synchronous throughput typically limited to hundreds of messages per second per thread while thread pool sizing constrains concurrent publishing capacity requiring careful resource allocation and capacity planning. Asynchronous throughput can reach thousands to tens of thousands of messages per second while callback execution overhead and memory usage for pending operations limit practical concurrent operation count. Timeout configuration affects both patterns while network characteristics determine practical performance limits and reliability characteristics for different deployment environments.

Default Values Synchronous operations use underlying Kafka producer timeout defaults (typically 30 seconds) while async operations use KafkaTemplate callback coordination with default thread pool sizing. Error handling follows Spring exception translation patterns while timeout configuration inherits from producer client settings requiring explicit tuning for production scenarios.

Best Practices Choose synchronous sending for low-throughput scenarios requiring immediate error handling while using asynchronous sending for high-throughput applications with proper callback error handling and monitoring. Configure appropriate timeouts based on network characteristics and application requirements while implementing comprehensive error handling strategies for both sync and async patterns. Monitor publishing performance and error rates while tuning thread pool sizing and callback execution for optimal throughput and reliability characteristics in production deployments.

2.1.2 Callback handling

Definition KafkaTemplate callback handling provides asynchronous result processing through ListenableFutureCallback and SendCallback interfaces enabling non-blocking publishing with custom success and failure handling logic. Callback mechanisms coordinate with Spring's async infrastructure while providing access to publishing results, metadata, and exception information for comprehensive error handling and monitoring integration.

Key Highlights Callback interfaces provide onSuccess and onFailure methods with access to SendResult metadata including partition, offset, and timestamp information while exception handling provides detailed error information for retry and recovery strategies. Spring integration supports ListenableFuture patterns with callback chaining while maintaining compatibility with CompletableFuture and reactive programming models for modern async application patterns. Thread pool coordination ensures callback execution on appropriate threads while avoiding blocking producer client threads and maintaining optimal publishing performance characteristics.

Responsibility / Role Callback coordination manages asynchronous result processing while providing access to publishing metadata and error information for monitoring, logging, and retry logic implementation. Error handling coordination processes publishing failures while integrating with Spring's error handling infrastructure and retry mechanisms for comprehensive fault tolerance patterns. Thread management ensures

callback execution without blocking producer operations while coordinating with Spring's async thread pools and execution infrastructure for optimal resource utilization.

Underlying Data Structures / Mechanism Callback implementation uses functional interfaces with method references and lambda expressions while coordinating with underlying Kafka producer callback mechanisms and result processing. Thread pool coordination uses Spring's async infrastructure while ensuring callback execution on appropriate threads and maintaining producer client performance characteristics. Error propagation mechanisms provide exception translation while maintaining stack trace information and integration with Spring's exception handling and monitoring infrastructure.

Advantages Non-blocking callback processing enables high-throughput publishing while providing access to detailed result metadata for monitoring and debugging applications requiring comprehensive operational visibility. Flexible callback implementation through functional interfaces while Spring integration provides consistent programming model with other async components and reactive programming patterns. Error handling capabilities through dedicated failure callbacks while maintaining integration with Spring's retry and recovery mechanisms for production-grade fault tolerance.

Disadvantages / Trade-offs Callback complexity increases application logic while error handling becomes distributed across callback methods requiring careful coordination and potentially sophisticated error recovery strategies. Thread pool management overhead while callback execution coordination can affect performance under high load requiring careful thread pool sizing and callback optimization. Memory usage increases with pending callbacks while callback failures can cause silent errors requiring comprehensive error handling and monitoring for callback execution health.

Corner Cases Callback execution failures can cause silent errors while callback thread pool exhaustion can cause execution delays and potential memory leaks requiring comprehensive resource management and error handling strategies. Callback timing issues during application shutdown while Spring context termination can interfere with callback execution and completion requiring proper lifecycle coordination and graceful shutdown procedures. Exception handling within callbacks can cause secondary failures while callback coordination with Spring transactions may not behave as expected requiring careful transaction boundary management.

Limits / Boundaries Callback execution performance depends on thread pool sizing while concurrent callback count is limited by available system resources and memory allocation for pending operations. Callback complexity is constrained by functional interface limitations while error handling capabilities depend on Spring's exception translation and retry infrastructure. Maximum pending callback count depends on memory allocation while callback execution latency affects overall publishing performance and application responsiveness characteristics.

Default Values Callback thread pool uses Spring's async infrastructure defaults while error handling follows standard exception translation patterns with basic logging and error propagation. Callback timeout coordination inherits from underlying producer settings while thread pool sizing uses framework defaults requiring explicit configuration for production workload characteristics.

Best Practices Implement lightweight callback logic while avoiding blocking operations and complex processing that could affect callback execution performance and system resource utilization. Design comprehensive error handling strategies within callbacks while integrating with application monitoring and alerting systems for operational visibility and incident response. Configure appropriate thread pool sizing for

callback execution while monitoring callback performance and error rates ensuring optimal async publishing characteristics and application reliability.

2.2 Producer Configuration

2.2.1 Key/Value serializers

Definition Spring Kafka serializer configuration provides type-safe message serialization through configurable key and value serializers with support for primitive types, JSON serialization, Avro integration, and custom serialization logic. Serializer configuration integrates with Spring's type conversion system while supporting Schema Registry integration and complex object serialization for enterprise data integration patterns.

Key Highlights Built-in serializer support includes StringSerializer, IntegerSerializer, and ByteArraySerializer while JsonSerializer provides automatic object serialization with Jackson integration and configurable type mapping. Schema Registry integration enables Avro serialization with automatic schema management while custom serializers support application-specific serialization requirements and performance optimization. Type safety coordination through generics while Spring Boot auto-configuration provides sensible defaults with override capabilities for production optimization.

Responsibility / Role Serialization coordination manages type conversion from Java objects to byte arrays while maintaining type safety and performance characteristics for various data types and serialization formats. Configuration management handles serializer selection and parameterization while integrating with Spring property binding and environment-specific configuration for deployment flexibility. Error handling manages serialization failures while providing comprehensive error information and integration with Spring's exception translation infrastructure for reliable message publishing.

Underlying Data Structures / Mechanism Serializer implementation uses Kafka's Serializer interface while Spring configuration provides bean-based serializer management with dependency injection and lifecycle coordination. Type resolution uses Spring's generic type handling while JsonSerializer leverages Jackson's ObjectMapper with configurable serialization features and type information handling. Schema Registry integration uses client libraries with schema caching while custom serializers provide direct byte array generation with performance optimization and error handling capabilities.

Advantages Type-safe serialization through generics while automatic JSON serialization eliminates manual object-to-byte conversion for common use cases and rapid development scenarios. Schema Registry integration provides schema evolution support while custom serializers enable performance optimization and application-specific serialization requirements. Spring configuration management enables environment-specific serializer selection while maintaining type safety and comprehensive error handling for production deployment scenarios.

Disadvantages / Trade-offs Serialization overhead can affect publishing performance while JSON serialization may not provide optimal size or performance compared to binary formats requiring careful evaluation for high-throughput scenarios. Schema Registry dependency increases infrastructure complexity while custom serializers require maintenance and testing overhead affecting development velocity and operational complexity. Type erasure limitations with generics while complex object serialization can cause memory allocation and garbage collection pressure requiring performance optimization and monitoring.

Corner Cases Serialization failures can cause publishing errors while type mismatch issues can cause runtime exceptions requiring comprehensive error handling and type validation strategies. Schema evolution conflicts

can cause serialization failures while JSON serialization of complex types may not preserve object semantics requiring careful object design and serialization testing. Custom serializer bugs can cause data corruption while serializer state management can cause thread safety issues requiring careful implementation and testing procedures.

Limits / Boundaries Serialization performance varies significantly between serializer types while JSON serialization typically provides lower throughput compared to binary formats requiring performance testing and optimization. Maximum object size for serialization depends on serializer implementation while Schema Registry has limits on schema size and complexity affecting data model design. Custom serializer complexity is bounded by implementation effort while error handling capabilities depend on serializer design and integration with Spring's exception handling infrastructure.

Default Values Spring Kafka uses StringSerializer for both keys and values by default while JsonSerializer configuration provides reasonable Jackson defaults with type information handling. Schema Registry serializers require explicit configuration while custom serializers need complete implementation and configuration requiring explicit setup for production deployments.

Best Practices Choose appropriate serializers based on performance requirements and data characteristics while implementing comprehensive error handling for serialization failures and type conversion issues. Configure Schema Registry integration for enterprise data management while implementing custom serializers for performance-critical applications requiring optimization and specialized data formats. Monitor serialization performance and error rates while implementing appropriate type validation and error recovery strategies ensuring reliable message publishing and data integrity across application evolution.

2.2.2 Custom partitioners

Definition Custom partitioners in Spring Kafka enable application-specific partition assignment logic beyond default key hashing, providing control over message distribution across partitions for load balancing, data locality, and business logic requirements. Partitioner implementation integrates with Spring configuration management while supporting dynamic partition assignment and sophisticated routing strategies for enterprise messaging patterns.

Key Highlights Custom partitioner implementation uses Kafka's Partitioner interface while Spring configuration enables bean-based partitioner management with dependency injection and lifecycle coordination for complex partitioning logic. Business logic integration enables partition assignment based on message content, routing keys, or external factors while maintaining partition count awareness and dynamic partition handling. Performance optimization through efficient partition calculation while maintaining consistent partition assignment for ordering guarantees and consumer coordination requirements.

Responsibility / Role Partition assignment coordination manages message routing across available partitions while considering load balancing, data locality, and business logic requirements for optimal message distribution and consumer utilization. Configuration management handles partitioner selection and parameterization while integrating with Spring property binding and environment-specific configuration for deployment flexibility and operational management. Error handling manages partition calculation failures while providing fallback strategies and integration with Spring's exception handling infrastructure for reliable message publishing.

Underlying Data Structures / Mechanism Partitioner implementation receives message metadata including topic, key, value, and cluster information while calculating appropriate partition assignment using custom

business logic and mathematical algorithms. Spring configuration uses bean definition with dependency injection while partitioner lifecycle management coordinates with producer client initialization and topic metadata updates. Performance optimization includes partition calculation caching while maintaining thread safety and concurrent access patterns for high-throughput publishing scenarios.

Advantages Fine-grained control over partition assignment enables optimization for specific business requirements including data locality, load balancing, and consumer coordination patterns. Spring integration provides dependency injection and configuration management while custom logic can consider external factors and business rules for sophisticated message routing strategies. Performance optimization through efficient partition calculation while maintaining consistency guarantees and integration with Kafka's ordering and consumer coordination semantics.

Disadvantages / Trade-offs Implementation complexity increases with custom partitioning logic while debugging partition assignment issues can be challenging requiring comprehensive testing and validation procedures. Performance overhead from custom calculation while poorly implemented partitioners can cause hotspots and uneven load distribution affecting consumer performance and cluster utilization. Maintenance overhead for custom logic while partition count changes require partitioner updates and potential data migration affecting operational procedures and application evolution.

Corner Cases Partition count changes can break custom partitioning logic while partitioner failures can cause publishing errors requiring comprehensive error handling and fallback strategies for operational reliability. Thread safety issues with stateful partitioners while partition calculation exceptions can cause message publishing failures requiring careful implementation and error recovery procedures. Hot partition scenarios from biased partitioning while consumer rebalancing can be affected by partition assignment patterns requiring careful partition strategy design and monitoring.

Limits / Boundaries Partition calculation performance affects overall publishing throughput while complex partitioning logic can become bottleneck requiring optimization and potentially caching strategies for high-throughput scenarios. Maximum partition count depends on cluster configuration while partitioner complexity is bounded by implementation effort and performance requirements. Custom logic sophistication limited by available message metadata while external system integration can affect partitioning performance and reliability characteristics.

Default Values Kafka uses default hash-based partitioning while Spring Kafka inherits these defaults with override capability through configuration properties and custom bean definitions. Partition assignment uses round-robin for null keys while hash-based assignment uses murmur2 hashing algorithm for consistent partition distribution.

Best Practices Design custom partitioners with partition count scalability in mind while implementing efficient calculation algorithms that avoid hotspots and uneven load distribution across consumer instances. Implement comprehensive error handling and fallback strategies while monitoring partition distribution and consumer utilization for optimal load balancing and performance characteristics. Test partition assignment logic thoroughly while considering partition count changes and consumer scaling scenarios ensuring reliable message distribution and consumer coordination across application evolution and operational scaling requirements.

2.3 Transactional Producers

2.3.1 Enabling transactions

Definition Transactional producers in Spring Kafka provide exactly-once semantics across multiple partitions and topics through coordinated transaction management with automatic producer ID allocation and transaction coordinator integration. Transaction enablement requires configuration of transactional.id property while Spring integration provides declarative transaction support through @Transactional annotation and transaction template patterns.

Key Highlights Transaction configuration requires unique transactional.id per producer instance while Spring Boot auto-configuration simplifies transaction enablement through property-based configuration and automatic bean setup. Declarative transaction support through @Transactional annotation while transaction boundaries coordinate with database operations and other transactional resources for comprehensive exactly-once processing patterns. Transaction coordinator integration manages distributed transaction state while providing automatic recovery and zombie producer detection for reliable exactly-once semantics across application restarts and failure scenarios.

Responsibility / Role Transaction coordination manages producer session state and transaction boundaries while integrating with Spring's transaction management infrastructure for declarative transaction handling and resource coordination. Producer ID allocation and session management coordinates with Kafka transaction coordinators while providing automatic recovery and zombie detection for reliable exactly-once processing across distributed system boundaries. Error handling manages transaction failures while providing rollback capabilities and integration with Spring's transaction infrastructure for consistent error recovery and resource cleanup.

Underlying Data Structures / Mechanism Transactional producer configuration uses unique producer IDs with coordinator assignment while transaction state management coordinates with Spring's transaction synchronization infrastructure and resource managers. Transaction boundary coordination uses begin/commit/abort operations while integrating with Spring's transaction template and declarative transaction management for consistent resource coordination. Coordinator communication manages transaction markers and participant coordination while providing automatic recovery and cleanup for distributed transaction consistency.

Advantages Exactly-once semantics eliminate duplicate processing concerns while Spring integration provides declarative transaction management with consistent programming model across different transactional resources and enterprise integration patterns. Automatic recovery and zombie detection provide operational resilience while coordinated transactions across Kafka and databases enable reliable business process implementation with strong consistency guarantees. Transaction rollback capabilities enable error recovery while maintaining data consistency and business logic integrity during failure scenarios.

Disadvantages / Trade-offs Significant performance overhead typically reducing throughput by 30-50% while increasing latency due to transaction coordination protocols and distributed consensus requirements affecting application scalability characteristics. Operational complexity increases substantially including transaction coordinator capacity planning while error handling becomes more complex requiring sophisticated recovery strategies and monitoring procedures. Resource utilization increases with transaction state management while concurrent transaction limits affect application throughput and scaling characteristics requiring careful capacity planning.

Corner Cases Transaction coordinator failures can cause transaction unavailability while producer session conflicts can cause authentication and coordination issues requiring comprehensive error handling and recovery procedures. Transaction timeout coordination can cause automatic rollbacks while network partitions

can affect transaction completion and coordinator availability requiring timeout tuning and operational monitoring. Spring transaction boundary coordination with async operations can cause unexpected behavior while transaction propagation across different thread contexts may not work as expected requiring careful transaction design.

Limits / Boundaries Transaction timeout ranges from 1 second to 15 minutes while coordinator capacity typically supports thousands of concurrent transactions depending on cluster configuration and hardware characteristics. Maximum transaction participants include all affected partitions while transaction state storage affects coordinator memory and disk utilization requiring capacity planning for high-transaction-rate applications. Producer session limits depend on coordinator resources while transaction throughput is constrained by coordination overhead and network characteristics affecting application performance and scaling.

Default Values Transactional producers require explicit transactional.id configuration while transaction timeout defaults to 60 seconds with coordinator selection using hash-based assignment. Spring transaction propagation defaults to REQUIRED while isolation level follows Spring transaction management defaults requiring explicit configuration for production transaction patterns and business requirements.

Best Practices Configure unique transactional.id per producer instance while implementing comprehensive error handling for transaction failures and coordinator unavailability scenarios affecting application reliability and data consistency. Design transaction boundaries carefully while coordinating with Spring's transaction management and avoiding long-running transactions that can cause coordinator resource exhaustion. Monitor transaction performance and coordinator health while implementing appropriate timeout and retry strategies ensuring reliable exactly-once processing and operational resilience for business-critical applications requiring strong consistency guarantees.

2.3.2 Idempotent producer

Definition Idempotent producers in Spring Kafka eliminate duplicate messages within partition boundaries through producer ID and sequence number coordination, providing exactly-once semantics for individual partitions without the complexity and overhead of full distributed transactions. Idempotency enablement requires minimal configuration while providing automatic duplicate detection and prevention with significantly lower performance overhead compared to full transactional producers.

Key Highlights Producer ID allocation provides unique identifier per producer session while sequence number tracking ensures duplicate detection and prevention within partition boundaries with automatic retry coordination. Spring Boot auto-configuration enables idempotency through simple property configuration while maintaining compatibility with existing producer patterns and minimal code changes for application upgrade. Performance overhead is minimal typically less than 5% compared to non-idempotent producers while providing exactly-once guarantees within partition scope and automatic duplicate prevention during retry scenarios.

Responsibility / Role Duplicate detection manages producer ID and sequence number coordination while preventing duplicate message delivery within partition boundaries through broker-side validation and sequence gap detection. Session management coordinates producer ID allocation and renewal while providing automatic recovery across application restarts and network partition scenarios with minimal operational overhead. Error handling manages sequence validation failures while providing retry coordination and automatic recovery for transient failures and network issues affecting message publishing reliability.

Underlying Data Structures / Mechanism Producer ID allocation uses 64-bit identifiers with broker coordination while sequence numbers use 32-bit per-partition counters for duplicate detection with automatic overflow handling and session renewal. Broker-side validation stores recent sequence numbers in memory while providing efficient duplicate detection without persistent storage requirements affecting broker performance and scalability. Session coordination manages producer ID lifecycle while automatic renewal prevents session expiration and maintains continuous idempotency guarantees across long-running applications.

Advantages Exactly-once semantics within partition boundaries while maintaining minimal performance overhead and operational complexity compared to full distributed transactions making it suitable for most reliability requirements. Automatic duplicate prevention eliminates application-level deduplication logic while Spring integration provides seamless enablement through configuration properties and auto-configuration. Simple configuration and operation while providing significant reliability improvements for retry scenarios and network partition recovery without complex coordination or infrastructure requirements.

Disadvantages / Trade-offs Partition-scope limitation prevents cross-partition exactly-once guarantees while producer ID exhaustion after 2 billion messages requires session renewal with potential temporary unavailability affecting long-running high-throughput applications. Sequence number coordination overhead while broker memory usage for sequence tracking affects cluster capacity and performance characteristics requiring monitoring and capacity planning. Limited transaction scope compared to full transactional producers while some use cases require cross-partition atomicity necessitating more complex transaction management approaches.

Corner Cases Producer ID conflicts during session renewal while broker failures can cause temporary sequence validation issues requiring producer restart and potential duplicate detection during recovery scenarios. Sequence number gaps trigger producer errors while network partitions can cause session timeout and ID renewal affecting continuous operation and requiring application-level error handling. Partition count changes can affect sequence number coordination while producer session management during cluster rebalancing can cause temporary availability issues requiring operational coordination.

Limits / Boundaries Producer session supports 2 billion messages per partition while 15-minute session timeout requires periodic activity or explicit session management for long-running idle applications. Memory usage for sequence tracking scales with active producer count while broker capacity typically supports thousands of concurrent idempotent producers depending on hardware and configuration characteristics. Performance impact is minimal but scales with message rate while sequence validation overhead increases with concurrent producer count requiring capacity planning for high-throughput scenarios.

Default Values Idempotent producers are disabled by default requiring explicit configuration through enable.idempotence property while producer ID timeout defaults to 15 minutes with automatic renewal coordination. Sequence number management uses framework defaults while error handling follows standard retry patterns with exponential backoff and automatic recovery for transient failures.

Best Practices Enable idempotent producers for all production applications requiring reliability while monitoring producer ID allocation and session management for long-running applications and high-throughput scenarios. Implement comprehensive error handling for producer ID renewal and sequence validation failures while designing applications to handle temporary unavailability during session renewal procedures. Monitor sequence tracking and broker memory usage while implementing appropriate session

management for idle producers ensuring continuous idempotency guarantees and optimal resource utilization across application lifecycle and operational requirements.