Spring Kafka Advanced Features: Part 2 - Kafka Streams & Connect Integration

Continuation of the comprehensive Spring Kafka Advanced Features guide covering Kafka Streams integration with Spring and Kafka Connect setup for enterprise data pipelines.

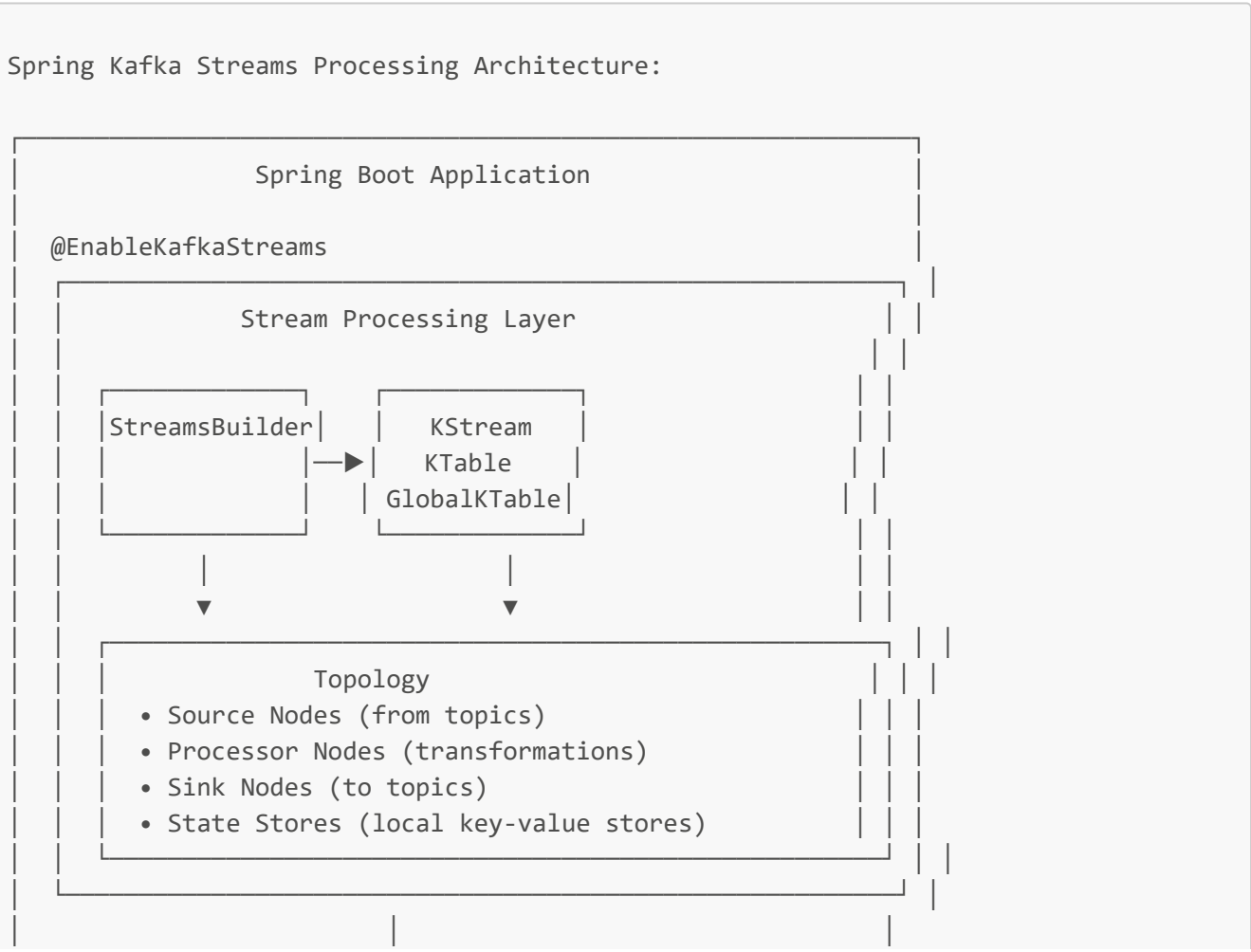
Kafka Streams Integration with Spring

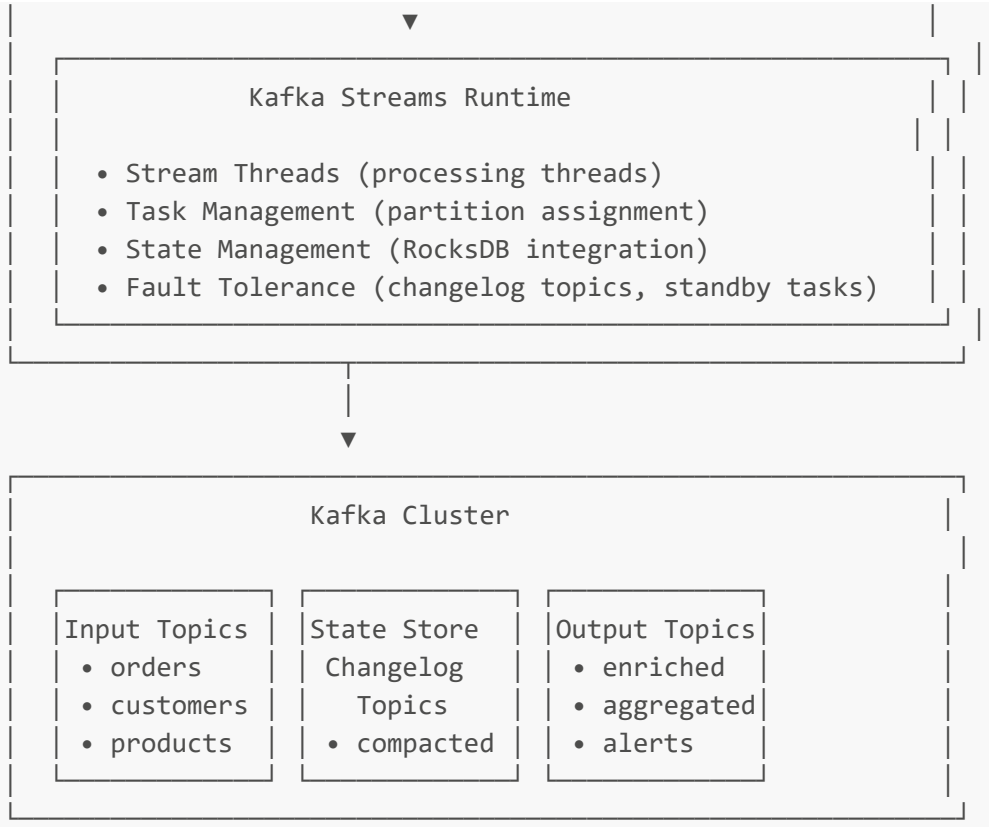
Simple Explanation: Kafka Streams Integration with Spring provides a declarative way to build stream processing applications using Spring Boot's auto-configuration. It enables real-time data transformation, aggregation, and analysis with minimal boilerplate code while maintaining Spring's dependency injection and configuration management.

What Problem It Solves:

- **Real-time Data Processing**: Transform and analyze data streams as they flow through Kafka
- **Stateful Stream Processing**: Maintain state across stream operations (joins, aggregations, windowing)
- **Complex Event Processing**: Detect patterns and correlations in event streams
- **Data Enrichment**: Join streams with reference data and external sources
- **Stream Analytics**: Calculate metrics, alerts, and insights from streaming data

Kafka Streams Architecture:






```
Stream Processing Flow:
1. Input topics feed data into KStreams/KTables
2. StreamsBuilder defines processing topology
3. Transformations: filter, map, join, aggregate, window
4. State stores maintain local state for stateful operations
5. Output streams written to destination topics
6. Changelog topics provide fault tolerance
7. Stream threads process partitions independently
```

Complete Kafka Streams Integration Implementation

```java
import org.apache.kafka.streams.StreamsBuilder;
import org.apache.kafka.streams.StreamsConfig;
import org.apache.kafka.streams.kstream.*;
import org.apache.kafka.streams.state.Stores;
import org.springframework.kafka.annotation.EnableKafkaStreams;
import org.springframework.kafka.config.KafkaStreamsConfiguration;

/**
 * Comprehensive Kafka Streams configuration with Spring Boot
 */
@Configuration
@EnableKafka
@EnableKafkaStreams
@lombok.extern.slf4j.Slf4j
public class KafkaStreamsConfiguration {

    @Value("${kafka.streams.application-id:advanced-streams-app}")
```

```
    private String applicationId;

    @Value("${kafka.streams.bootstrap-servers:localhost:9092}")
    private String bootstrapServers;

    @Value("${kafka.streams.num-stream-threads:3}")
    private Integer numStreamThreads;

    /**
     * Kafka Streams configuration bean
     */
    @Bean(name =
KafkaStreamsDefaultConfiguration.DEFAULT_STREAMS_CONFIG_BEAN_NAME)
    public KafkaStreamsConfiguration kStreamsConfig() {

        Map<String, Object> props = new HashMap<>();

        // Basic configuration
        props.put(StreamsConfig.APPLICATION_ID_CONFIG, applicationId);
        props.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, bootstrapServers);
        props.put(StreamsConfig.NUM_STREAM_THREADS_CONFIG, numStreamThreads);

        // Serialization configuration
        props.put(StreamsConfig.DEFAULT_KEY_SERDE_CLASS_CONFIG,
Serdes.String().getClass());
        props.put(StreamsConfig.DEFAULT_VALUE_SERDE_CLASS_CONFIG,
Serdes.String().getClass());

        // Processing configuration
        props.put(StreamsConfig.PROCESSING_GUARANTEE_CONFIG,
StreamsConfig.EXACTLY_ONCE_V2);
        props.put(StreamsConfig.COMMIT_INTERVAL_MS_CONFIG, 1000);
        props.put(StreamsConfig.CACHE_MAX_BYTES_BUFFERING_CONFIG, 10 * 1024 *
1024); // 10MB

        // State store configuration
        props.put(StreamsConfig.STATE_DIR_CONFIG, "/tmp/kafka-streams");
        props.put(StreamsConfig.REPLICATION_FACTOR_CONFIG, 3);

        // Performance tuning
        props.put(StreamsConfig.POLL_MS_CONFIG, 100);
        props.put(StreamsConfig.MAX_TASK_IDLE_MS_CONFIG, 0);

        // Error handling

props.put(StreamsConfig.DEFAULT_DESERIALIZATION_EXCEPTION_HANDLER_CLASS_CONFIG,
            LogAndContinueExceptionHandler.class);
        props.put(StreamsConfig.DEFAULT_PRODUCTION_EXCEPTION_HANDLER_CLASS_CONFIG,
            DefaultProductionExceptionHandler.class);

        log.info("Configured Kafka Streams with applicationId: {}, threads: {}",
            applicationId, numStreamThreads);

        return new KafkaStreamsConfiguration(props);
```

```
    }

    /**
     * Custom JSON Serde for complex objects
     */
    @Bean
    public Serde<OrderEvent> orderEventSerde() {
        return Serdes.serdeFrom(new JsonSerializer<>(), new JsonDeserializer<>
(OrderEvent.class));
    }

    @Bean
    public Serde<CustomerEvent> customerEventSerde() {
        return Serdes.serdeFrom(new JsonSerializer<>(), new JsonDeserializer<>
(CustomerEvent.class));
    }

    @Bean
    public Serde<EnrichedOrder> enrichedOrderSerde() {
        return Serdes.serdeFrom(new JsonSerializer<>(), new JsonDeserializer<>
(EnrichedOrder.class));
    }

    @Bean
    public Serde<OrderAggregate> orderAggregateSerde() {
        return Serdes.serdeFrom(new JsonSerializer<>(), new JsonDeserializer<>
(OrderAggregate.class));
    }

    /**
     * Streams builder factory customizer
     */
    @Bean
    public StreamsBuilderFactoryBeanConfigurer
streamsBuilderFactoryBeanConfigurer() {
        return factoryBean -> {
            factoryBean.setStateListener((newState, oldState) -> {
                log.info("Kafka Streams state transition: {} -> {}", oldState,
newState);

                // Add custom state management logic here
                if (newState == KafkaStreams.State.RUNNING) {
                    log.info("Kafka Streams application is now running");
                } else if (newState == KafkaStreams.State.ERROR) {
                    log.error("Kafka Streams application encountered an error");
                }
            });

            factoryBean.setUncaughtExceptionHandler((thread, exception) -> {
                log.error("Uncaught exception in stream thread: {}",
thread.getName(), exception);
                return
StreamsUncaughtExceptionHandler.StreamThreadExceptionResponse.REPLACE_THREAD;
            });
```

```
        };
    }
}

/**
 * Order Processing Stream - Basic transformations and filtering
 */
@Component
@lombok.extern.slf4j.Slf4j
public class OrderProcessingStream {

    @Autowired
    private Serde<OrderEvent> orderEventSerde;

    @Autowired
    private Serde<EnrichedOrder> enrichedOrderSerde;

    @Bean
    public KStream<String, OrderEvent> orderProcessingTopology(StreamsBuilder
streamsBuilder) {

        log.info("Setting up order processing topology");

        // Source: Read from order events topic
        KStream<String, OrderEvent> orderStream = streamsBuilder
            .stream("order-events", Consumed.with(Serdes.String(),
orderEventSerde))
            .peek((key, value) -> log.debug("Processing order: {}",
value.getOrderId()));

        // Filter: Only process orders above minimum amount
        KStream<String, OrderEvent> validOrders = orderStream
            .filter((key, order) -> order.getAmount().compareTo(new
BigDecimal("10.00")) > 0)
            .peek((key, value) -> log.debug("Valid order: {}",
value.getOrderId()));

        // Transform: Add processing timestamp and enrich basic data
        KStream<String, EnrichedOrder> enrichedOrders = validOrders
            .mapValues((readOnlyKey, order) -> {

                EnrichedOrder enriched = new EnrichedOrder();
                enriched.setOrderId(order.getOrderId());
                enriched.setCustomerId(order.getCustomerId());
                enriched.setAmount(order.getAmount());
                enriched.setItems(order.getItems());
                enriched.setOriginalTimestamp(order.getTimestamp());
                enriched.setProcessingTimestamp(Instant.now());
                enriched.setProcessingRegion(System.getProperty("region",
"default"));
                enriched.setStatus("PROCESSED");

                return enriched;
            })
```

```
            .peek((key, value) -> log.debug("Enriched order: {}",
value.getOrderId()));

        // Branch: Split orders by amount for different processing paths
        Map<String, KStream<String, EnrichedOrder>> branches =
enrichedOrders.split(Named.as("order-amount-"))
            .branch((key, order) -> order.getAmount().compareTo(new
BigDecimal("1000.00")) >= 0,
                    Branched.as("high-value"))
            .branch((key, order) -> order.getAmount().compareTo(new
BigDecimal("100.00")) >= 0,
                    Branched.as("medium-value"))
            .defaultBranch(Branched.as("low-value"));

        // High-value orders: Additional validation and alerts
        branches.get("order-amount-high-value")
            .mapValues(order -> {
                order.setRiskLevel("HIGH");
                order.setRequiresApproval(true);
                return order;
            })
            .to("high-value-orders", Produced.with(Serdes.String(),
enrichedOrderSerde));

        // Medium-value orders: Standard processing
        branches.get("order-amount-medium-value")
            .mapValues(order -> {
                order.setRiskLevel("MEDIUM");
                return order;
            })
            .to("medium-value-orders", Produced.with(Serdes.String(),
enrichedOrderSerde));

        // Low-value orders: Fast-track processing
        branches.get("order-amount-low-value")
            .mapValues(order -> {
                order.setRiskLevel("LOW");
                order.setFastTrack(true);
                return order;
            })
            .to("low-value-orders", Produced.with(Serdes.String(),
enrichedOrderSerde));

        // Write all enriched orders to main output topic
        enrichedOrders.to("enriched-orders", Produced.with(Serdes.String(),
enrichedOrderSerde));

        log.info("Order processing topology configured successfully");

        return orderStream;
    }
}

/**
```

```
 * Customer Order Analytics Stream - Aggregations and windowing
 */
@Component
@lombok.extern.slf4j.Slf4j
public class CustomerAnalyticsStream {

    @Autowired
    private Serde<OrderEvent> orderEventSerde;

    @Autowired
    private Serde<OrderAggregate> orderAggregateSerde;

    @Bean
    public KTable<String, OrderAggregate> customerAnalyticsTopology(StreamsBuilder
streamsBuilder) {

        log.info("Setting up customer analytics topology");

        // Source: Read from order events
        KStream<String, OrderEvent> orderStream = streamsBuilder
            .stream("order-events", Consumed.with(Serdes.String(),
orderEventSerde));

        // Group by customer ID for aggregation
        KGroupedStream<String, OrderEvent> ordersByCustomer = orderStream
            .selectKey((key, order) -> order.getCustomerId())
            .groupByKey(Grouped.with(Serdes.String(), orderEventSerde));

        // Tumbling window aggregation: Customer order statistics per hour
        TimeWindows tumblingWindow =
TimeWindows.ofSizeWithNoGrace(Duration.ofHours(1));

        KTable<Windowed<String>, OrderAggregate> hourlyCustomerStats =
ordersByCustomer
            .windowedBy(tumblingWindow)
            .aggregate(
                // Initializer
                () -> OrderAggregate.builder()
                    .orderCount(0L)
                    .totalAmount(BigDecimal.ZERO)
                    .averageAmount(BigDecimal.ZERO)
                    .minAmount(null)
                    .maxAmount(null)
                    .build(),

                // Aggregator
                (customerId, order, aggregate) -> {
                    long newCount = aggregate.getOrderCount() + 1;
                    BigDecimal newTotal =
aggregate.getTotalAmount().add(order.getAmount());
                    BigDecimal newAverage =
newTotal.divide(BigDecimal.valueOf(newCount), 2, RoundingMode.HALF_UP);

                    BigDecimal newMin = aggregate.getMinAmount() != null ?
```

```
                        aggregate.getMinAmount().min(order.getAmount()) :
order.getAmount();
                    BigDecimal newMax = aggregate.getMaxAmount() != null ?
                        aggregate.getMaxAmount().max(order.getAmount()) :
order.getAmount();

                    return aggregate.toBuilder()
                        .customerId(customerId)
                        .orderCount(newCount)
                        .totalAmount(newTotal)
                        .averageAmount(newAverage)
                        .minAmount(newMin)
                        .maxAmount(newMax)
                        .lastUpdated(Instant.now())
                        .build();
                },

                // Materialized store
                Materialized.<String, OrderAggregate, WindowStore<Bytes,
byte[]>>as("hourly-customer-stats")
                    .withKeySerde(Serdes.String())
                    .withValueSerde(orderAggregateSerde)
            )

.suppress(Suppressed.of(Suppressed.BufferConfig.unbounded()).untilWindowCloses(Sup
pressed.StrictBufferConfig.unbounded()));

        // Convert windowed results to regular KTable for output
        KTable<String, OrderAggregate> customerStats = hourlyCustomerStats
            .toStream()
            .selectKey((windowedKey, aggregate) -> windowedKey.key())
            .toTable(Materialized.<String, OrderAggregate, KeyValueStore<Bytes,
byte[]>>as("customer-stats")
                .withKeySerde(Serdes.String())
                .withValueSerde(orderAggregateSerde));

        // Output to topic
        customerStats.toStream().to("customer-analytics",
Produced.with(Serdes.String(), orderAggregateSerde));

        // Session window aggregation: Customer session analysis
        SessionWindows sessionWindow =
SessionWindows.ofInactivityGapWithNoGrace(Duration.ofMinutes(30));

        KTable<Windowed<String>, OrderAggregate> customerSessions =
ordersByCustomer
            .windowedBy(sessionWindow)
            .aggregate(
                () ->
OrderAggregate.builder().orderCount(0L).totalAmount(BigDecimal.ZERO).build(),
                (customerId, order, aggregate) -> {
                    return aggregate.toBuilder()
                        .customerId(customerId)
                        .orderCount(aggregate.getOrderCount() + 1)
```

```
.totalAmount(aggregate.getTotalAmount().add(order.getAmount()))
                        .sessionStart(aggregate.getSessionStart() != null ?
                            aggregate.getSessionStart() : order.getTimestamp())
                        .sessionEnd(order.getTimestamp())
                        .lastUpdated(Instant.now())
                        .build();
                },
                (customerId, leftAggregate, rightAggregate) -> {
                    // Session merge logic
                    return leftAggregate.toBuilder()
                        .orderCount(leftAggregate.getOrderCount() +
rightAggregate.getOrderCount())

.totalAmount(leftAggregate.getTotalAmount().add(rightAggregate.getTotalAmount()))

.sessionStart(leftAggregate.getSessionStart().isBefore(rightAggregate.getSessionSt
art()) ?
                            leftAggregate.getSessionStart() :
rightAggregate.getSessionStart())

.sessionEnd(leftAggregate.getSessionEnd().isAfter(rightAggregate.getSessionEnd())
?
                            leftAggregate.getSessionEnd() :
rightAggregate.getSessionEnd())
                        .lastUpdated(Instant.now())
                        .build();
                },
                Materialized.<String, OrderAggregate, SessionStore<Bytes,
byte[]>>as("customer-sessions")
                    .withKeySerde(Serdes.String())
                    .withValueSerde(orderAggregateSerde)
            );

        // Output session data
        customerSessions.toStream()
            .selectKey((windowedKey, aggregate) -> windowedKey.key())
            .to("customer-sessions", Produced.with(Serdes.String(),
orderAggregateSerde));

        log.info("Customer analytics topology configured successfully");

        return customerStats;
    }
}

/**
 * Stream-Table Join Processing - Enrichment with reference data
 */
@Component
@lombok.extern.slf4j.Slf4j
public class StreamTableJoinProcessor {

    @Autowired
```

```
    private Serde<OrderEvent> orderEventSerde;

    @Autowired
    private Serde<CustomerEvent> customerEventSerde;

    @Autowired
    private Serde<EnrichedOrder> enrichedOrderSerde;

    @Bean
    public KStream<String, EnrichedOrder> streamTableJoinTopology(StreamsBuilder
streamsBuilder) {

        log.info("Setting up stream-table join topology");

        // Table: Customer reference data (compacted topic)
        KTable<String, CustomerEvent> customerTable = streamsBuilder
            .table("customer-events", Consumed.with(Serdes.String(),
customerEventSerde),
                   Materialized.<String, CustomerEvent, KeyValueStore<Bytes,
byte[]>>as("customer-store")
                       .withKeySerde(Serdes.String())
                       .withValueSerde(customerEventSerde));

        // Stream: Order events to enrich
        KStream<String, OrderEvent> orderStream = streamsBuilder
            .stream("order-events", Consumed.with(Serdes.String(),
orderEventSerde));

        // Join: Enrich orders with customer data
        KStream<String, EnrichedOrder> enrichedOrderStream = orderStream
            .selectKey((key, order) -> order.getCustomerId()) // Re-key by
customer ID
            .join(
                customerTable,
                (order, customer) -> {

                    log.debug("Joining order {} with customer {}",
                        order.getOrderId(), customer.getCustomerId());

                    EnrichedOrder enriched = new EnrichedOrder();

                    // Order data
                    enriched.setOrderId(order.getOrderId());
                    enriched.setCustomerId(order.getCustomerId());
                    enriched.setAmount(order.getAmount());
                    enriched.setItems(order.getItems());
                    enriched.setOriginalTimestamp(order.getTimestamp());

                    // Customer enrichment data
                    enriched.setCustomerName(customer.getName());
                    enriched.setCustomerEmail(customer.getEmail());
                    enriched.setCustomerSegment(customer.getSegment());

enriched.setCustomerLifetimeValue(customer.getLifetimeValue());
```

```
enriched.setCustomerRegistrationDate(customer.getRegistrationDate());

                    // Derived fields
                    enriched.setProcessingTimestamp(Instant.now());

enriched.setIsVipCustomer(customer.getSegment().equals("VIP"));
                    enriched.setRiskLevel(calculateRiskLevel(order, customer));

                    return enriched;
                },
                Joined.with(Serdes.String(), orderEventSerde, customerEventSerde)
            );

        // Filter out orders for inactive customers
        KStream<String, EnrichedOrder> activeCustomerOrders = enrichedOrderStream
            .filter((key, enrichedOrder) ->
!"INACTIVE".equals(enrichedOrder.getCustomerSegment()));

        // Write enriched orders to output topic
        activeCustomerOrders.to("enriched-orders-with-customer-data",
            Produced.with(Serdes.String(), enrichedOrderSerde));

        // Create alerts for high-risk orders
        KStream<String, EnrichedOrder> highRiskOrders = activeCustomerOrders
            .filter((key, enrichedOrder) ->
"HIGH".equals(enrichedOrder.getRiskLevel()));

        // Transform to alert format and send to alerts topic
        KStream<String, String> riskAlerts = highRiskOrders
            .mapValues(enrichedOrder -> {
                Map<String, Object> alert = new HashMap<>();
                alert.put("alertType", "HIGH_RISK_ORDER");
                alert.put("orderId", enrichedOrder.getOrderId());
                alert.put("customerId", enrichedOrder.getCustomerId());
                alert.put("customerName", enrichedOrder.getCustomerName());
                alert.put("amount", enrichedOrder.getAmount());
                alert.put("riskLevel", enrichedOrder.getRiskLevel());
                alert.put("timestamp", Instant.now().toString());

                try {
                    return new ObjectMapper().writeValueAsString(alert);
                } catch (Exception e) {
                    log.error("Error serializing alert", e);
                    return "{}";
                }
            });

        riskAlerts.to("risk-alerts", Produced.with(Serdes.String(),
Serdes.String()));

        log.info("Stream-table join topology configured successfully");

        return enrichedOrderStream;
```

```
    }

    private String calculateRiskLevel(OrderEvent order, CustomerEvent customer) {

        // Risk calculation logic based on order amount and customer history
        BigDecimal amount = order.getAmount();
        BigDecimal ltv = customer.getLifetimeValue();

        if (amount.compareTo(new BigDecimal("5000")) > 0 &&
            (ltv == null || ltv.compareTo(new BigDecimal("1000")) < 0)) {
            return "HIGH";
        } else if (amount.compareTo(new BigDecimal("1000")) > 0) {
            return "MEDIUM";
        } else {
            return "LOW";
        }
    }
}

/**
 * Interactive Queries Service - Query state stores
 */
@Service
@lombok.extern.slf4j.Slf4j
public class StreamsInteractiveQueryService {

    @Autowired
    private StreamsBuilderFactoryBean streamsBuilderFactoryBean;

    /**
     * Get customer statistics from state store
     */
    public Optional<OrderAggregate> getCustomerStats(String customerId) {

        try {
            KafkaStreams streams = streamsBuilderFactoryBean.getKafkaStreams();

            if (streams == null ||
!streams.state().equals(KafkaStreams.State.RUNNING)) {
                log.warn("Kafka Streams is not running, cannot query state
store");
                return Optional.empty();
            }

            ReadOnlyKeyValueStore<String, OrderAggregate> store = streams.store(
                StoreQueryParameters.fromNameAndType("customer-stats",
QueryableStoreTypes.keyValueStore())
            );

            OrderAggregate aggregate = store.get(customerId);
            return Optional.ofNullable(aggregate);

        } catch (Exception e) {
            log.error("Error querying customer stats for customerId: {}",
```

```
customerId, e);
            return Optional.empty();
        }
    }

    /**
     * Get all customer statistics
     */
    public Map<String, OrderAggregate> getAllCustomerStats() {

        try {
            KafkaStreams streams = streamsBuilderFactoryBean.getKafkaStreams();

            if (streams == null ||
!streams.state().equals(KafkaStreams.State.RUNNING)) {
                log.warn("Kafka Streams is not running, cannot query state
store");
                return Collections.emptyMap();
            }

            ReadOnlyKeyValueStore<String, OrderAggregate> store = streams.store(
                StoreQueryParameters.fromNameAndType("customer-stats",
QueryableStoreTypes.keyValueStore())
            );

            Map<String, OrderAggregate> results = new HashMap<>();

            try (KeyValueIterator<String, OrderAggregate> iterator = store.all())
{
                while (iterator.hasNext()) {
                    KeyValue<String, OrderAggregate> keyValue = iterator.next();
                    results.put(keyValue.key, keyValue.value);
                }
            }

            return results;

        } catch (Exception e) {
            log.error("Error querying all customer stats", e);
            return Collections.emptyMap();
        }
    }

    /**
     * Get customer statistics within range
     */
    public Map<String, OrderAggregate> getCustomerStatsInRange(String fromKey,
String toKey) {

        try {
            KafkaStreams streams = streamsBuilderFactoryBean.getKafkaStreams();

            if (streams == null ||
!streams.state().equals(KafkaStreams.State.RUNNING)) {
```

```
                return Collections.emptyMap();
            }

            ReadOnlyKeyValueStore<String, OrderAggregate> store = streams.store(
                StoreQueryParameters.fromNameAndType("customer-stats",
QueryableStoreTypes.keyValueStore())
            );

            Map<String, OrderAggregate> results = new HashMap<>();

            try (KeyValueIterator<String, OrderAggregate> iterator =
store.range(fromKey, toKey)) {
                while (iterator.hasNext()) {
                    KeyValue<String, OrderAggregate> keyValue = iterator.next();
                    results.put(keyValue.key, keyValue.value);
                }
            }

            return results;

        } catch (Exception e) {
            log.error("Error querying customer stats in range: {} to {}", fromKey,
toKey, e);
            return Collections.emptyMap();
        }
    }
}

/**
 * REST Controller exposing stream processing results
 */
@RestController
@RequestMapping("/api/streams")
@lombok.extern.slf4j.Slf4j
public class StreamsQueryController {

    @Autowired
    private StreamsInteractiveQueryService queryService;

    @GetMapping("/customers/{customerId}/stats")
    public ResponseEntity<OrderAggregate> getCustomerStats(@PathVariable String
customerId) {

        log.info("Querying customer stats for customerId: {}", customerId);

        Optional<OrderAggregate> stats =
queryService.getCustomerStats(customerId);

        if (stats.isPresent()) {
            return ResponseEntity.ok(stats.get());
        } else {
            return ResponseEntity.notFound().build();
        }
    }
```

```
    @GetMapping("/customers/stats")
    public ResponseEntity<Map<String, OrderAggregate>> getAllCustomerStats(
            @RequestParam(required = false) String fromKey,
            @RequestParam(required = false) String toKey) {

        log.info("Querying customer stats: fromKey={}, toKey={}", fromKey, toKey);

        Map<String, OrderAggregate> stats;

        if (fromKey != null && toKey != null) {
            stats = queryService.getCustomerStatsInRange(fromKey, toKey);
        } else {
            stats = queryService.getAllCustomerStats();
        }

        return ResponseEntity.ok(stats);
    }

    @GetMapping("/health")
    public ResponseEntity<Map<String, Object>> getStreamsHealth() {

        try {
            StreamsBuilderFactoryBean factoryBean =
queryService.getStreamsBuilderFactoryBean();
            KafkaStreams streams = factoryBean.getKafkaStreams();

            Map<String, Object> health = new HashMap<>();
            health.put("state", streams != null ? streams.state().name() :
"NOT_CREATED");
            health.put("timestamp", Instant.now().toString());

            if (streams != null) {
                health.put("threadMetadata", streams.metadataForLocalThreads());
            }

            return ResponseEntity.ok(health);

        } catch (Exception e) {
            log.error("Error checking streams health", e);

            Map<String, Object> errorHealth = Map.of(
                "state", "ERROR",
                "error", e.getMessage(),
                "timestamp", Instant.now().toString()
            );

            return
ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).body(errorHealth);
        }
    }
}

// Supporting data classes for Kafka Streams
```

```
@lombok.Data
@lombok.Builder
@lombok.NoArgsConstructor
@lombok.AllArgsConstructor
class OrderEvent {
    private String orderId;
    private String customerId;
    private BigDecimal amount;
    private List<String> items;
    private Instant timestamp;
    private String region;
    private Map<String, String> metadata;
}

@lombok.Data
@lombok.Builder
@lombok.NoArgsConstructor
@lombok.AllArgsConstructor
class CustomerEvent {
    private String customerId;
    private String name;
    private String email;
    private String segment;
    private BigDecimal lifetimeValue;
    private Instant registrationDate;
    private String status;
}

@lombok.Data
@lombok.Builder(toBuilder = true)
@lombok.NoArgsConstructor
@lombok.AllArgsConstructor
class EnrichedOrder {
    private String orderId;
    private String customerId;
    private BigDecimal amount;
    private List<String> items;
    private Instant originalTimestamp;
    private Instant processingTimestamp;
    private String processingRegion;
    private String status;
    private String riskLevel;
    private boolean requiresApproval;
    private boolean fastTrack;
    private boolean isVipCustomer;

    // Customer enrichment data
    private String customerName;
    private String customerEmail;
    private String customerSegment;
    private BigDecimal customerLifetimeValue;
    private Instant customerRegistrationDate;
}
```

```
@lombok.Data
@lombok.Builder(toBuilder = true)
@lombok.NoArgsConstructor
@lombok.AllArgsConstructor
class OrderAggregate {
    private String customerId;
    private Long orderCount;
    private BigDecimal totalAmount;
    private BigDecimal averageAmount;
    private BigDecimal minAmount;
    private BigDecimal maxAmount;
    private Instant sessionStart;
    private Instant sessionEnd;
    private Instant lastUpdated;
}
```

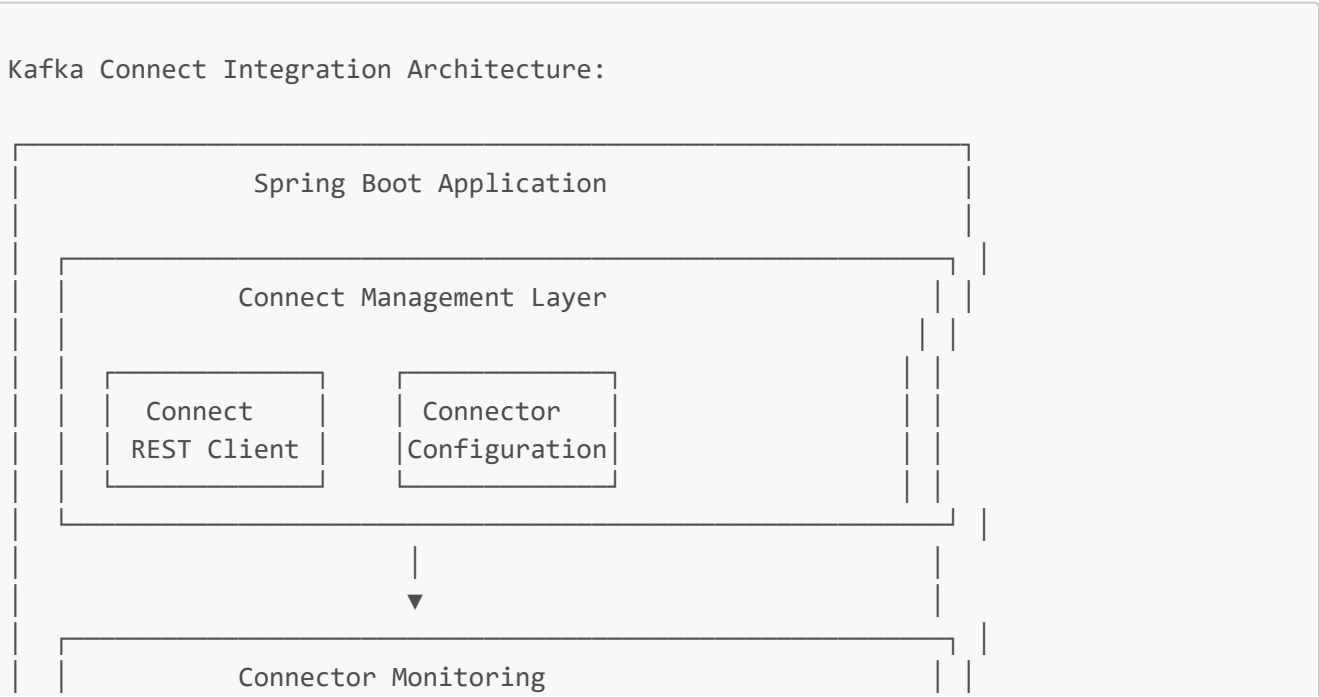
Kafka Connect Integration

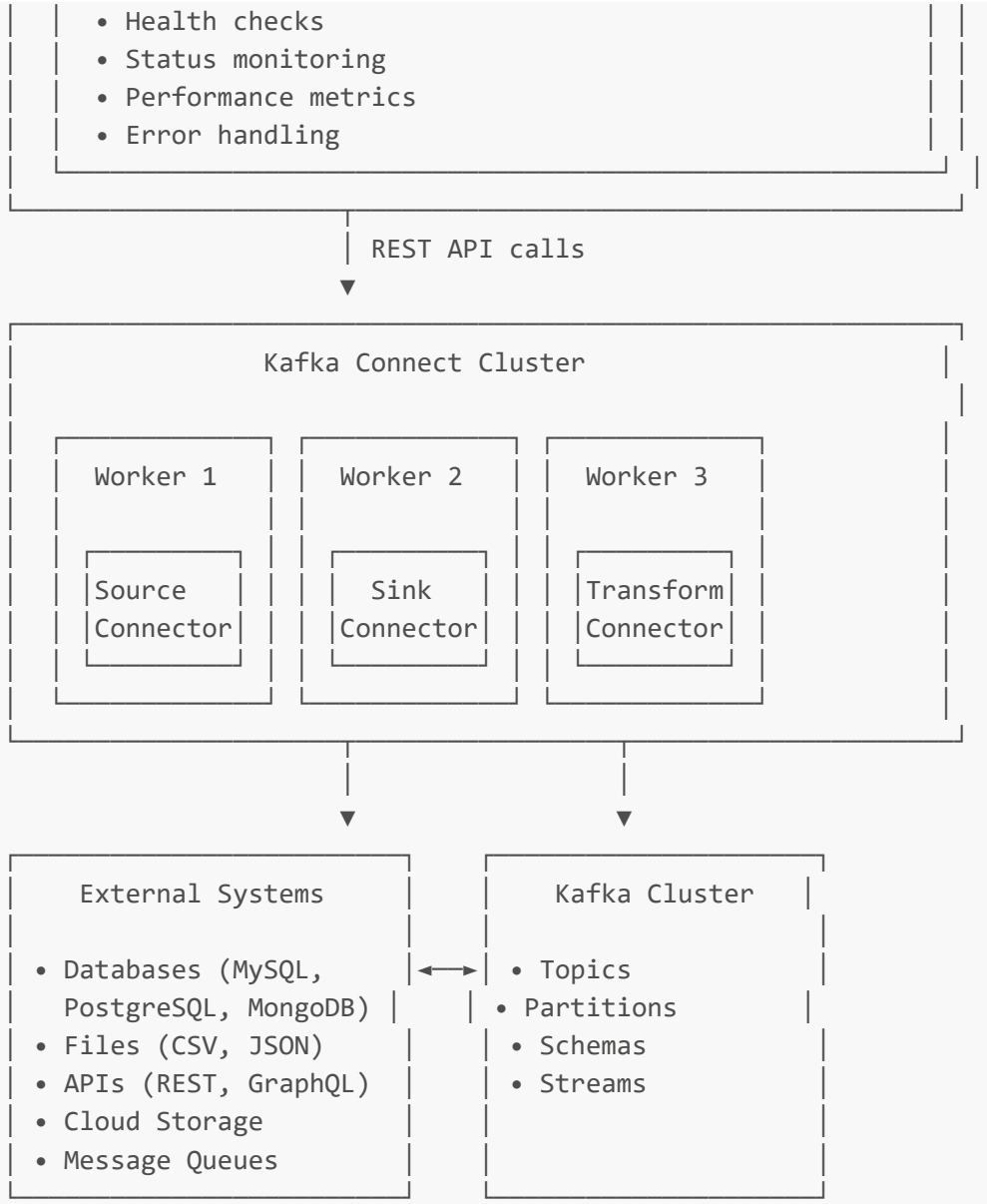
Simple Explanation: Kafka Connect Integration enables Spring applications to interact with Kafka Connect clusters for building data pipelines. It provides configuration management, connector deployment, monitoring capabilities, and seamless integration between external systems and Kafka topics through source and sink connectors.

What Problem It Solves:

- **Data Pipeline Management**: Automate data flow between Kafka and external systems
- **System Integration**: Connect databases, files, APIs, and cloud services to Kafka
- **Change Data Capture**: Stream database changes to Kafka in real-time
- **Data Lake Integration**: Move data between Kafka and data lakes/warehouses
- **Operational Simplification**: Manage connectors through Spring configuration

Kafka Connect Architecture:






```
Data Flow Examples:
Source Connectors: Database → Kafka Topic
Sink Connectors: Kafka Topic → External System
Transform Connectors: Topic A → Transform → Topic B
```

Complete Kafka Connect Integration Implementation

```
import org.springframework.web.client.RestTemplate;
import org.springframework.http.HttpEntity;
import org.springframework.http.HttpHeaders;
import org.springframework.http.HttpMethod;
import org.springframework.http.MediaType;

/**
 * Kafka Connect REST client for managing connectors
 */
@Service
@lombok.extern.slf4j.Slf4j
```

```
public class KafkaConnectService {

    @Value("${kafka.connect.url:http://localhost:8083}")
    private String connectUrl;

    private final RestTemplate restTemplate;
    private final ObjectMapper objectMapper;

    public KafkaConnectService() {
        this.restTemplate = new RestTemplate();
        this.objectMapper = new ObjectMapper();
    }

    /**
     * Get all connector names
     */
    public List<String> getConnectors() {

        try {
            String url = connectUrl + "/connectors";
            String[] connectors = restTemplate.getForObject(url, String[].class);

            return Arrays.asList(connectors != null ? connectors : new String[0]);

        } catch (Exception e) {
            log.error("Error getting connectors", e);
            return Collections.emptyList();
        }
    }

    /**
     * Get connector configuration
     */
    public Optional<Map<String, Object>> getConnectorConfig(String connectorName)
{

        try {
            String url = connectUrl + "/connectors/" + connectorName + "/config";

            @SuppressWarnings("unchecked")
            Map<String, Object> config = restTemplate.getForObject(url,
Map.class);

            return Optional.ofNullable(config);

        } catch (Exception e) {
            log.error("Error getting connector config: {}", connectorName, e);
            return Optional.empty();
        }
    }

    /**
     * Get connector status
     */
```

```
    public Optional<ConnectorStatus> getConnectorStatus(String connectorName) {

        try {
            String url = connectUrl + "/connectors/" + connectorName + "/status";
            ConnectorStatus status = restTemplate.getForObject(url,
ConnectorStatus.class);

            return Optional.ofNullable(status);

        } catch (Exception e) {
            log.error("Error getting connector status: {}", connectorName, e);
            return Optional.empty();
        }
    }

    /**
     * Create or update connector
     */
    public boolean createOrUpdateConnector(String connectorName, Map<String,
Object> config) {

        try {
            String url = connectUrl + "/connectors/" + connectorName + "/config";

            HttpHeaders headers = new HttpHeaders();
            headers.setContentType(MediaType.APPLICATION_JSON);

            String configJson = objectMapper.writeValueAsString(config);
            HttpEntity<String> entity = new HttpEntity<>(configJson, headers);

            restTemplate.exchange(url, HttpMethod.PUT, entity, String.class);

            log.info("Successfully created/updated connector: {}", connectorName);
            return true;

        } catch (Exception e) {
            log.error("Error creating/updating connector: {}", connectorName, e);
            return false;
        }
    }

    /**
     * Delete connector
     */
    public boolean deleteConnector(String connectorName) {

        try {
            String url = connectUrl + "/connectors/" + connectorName;
            restTemplate.delete(url);

            log.info("Successfully deleted connector: {}", connectorName);
            return true;

        } catch (Exception e) {
```

```
            log.error("Error deleting connector: {}", connectorName, e);
            return false;
        }
    }

    /**
     * Restart connector
     */
    public boolean restartConnector(String connectorName) {

        try {
            String url = connectUrl + "/connectors/" + connectorName + "/restart";
            restTemplate.postForObject(url, null, String.class);

            log.info("Successfully restarted connector: {}", connectorName);
            return true;

        } catch (Exception e) {
            log.error("Error restarting connector: {}", connectorName, e);
            return false;
        }
    }

    /**
     * Pause connector
     */
    public boolean pauseConnector(String connectorName) {

        try {
            String url = connectUrl + "/connectors/" + connectorName + "/pause";
            restTemplate.exchange(url, HttpMethod.PUT, null, String.class);

            log.info("Successfully paused connector: {}", connectorName);
            return true;

        } catch (Exception e) {
            log.error("Error pausing connector: {}", connectorName, e);
            return false;
        }
    }

    /**
     * Resume connector
     */
    public boolean resumeConnector(String connectorName) {

        try {
            String url = connectUrl + "/connectors/" + connectorName + "/resume";
            restTemplate.exchange(url, HttpMethod.PUT, null, String.class);

            log.info("Successfully resumed connector: {}", connectorName);
            return true;

        } catch (Exception e) {
```

```
            log.error("Error resuming connector: {}", connectorName, e);
            return false;
        }
    }

    /**
     * Validate connector configuration
     */
    public ConnectorValidationResult validateConnectorConfig(String
connectorClass, Map<String, String> config) {

        try {
            String url = connectUrl + "/connector-plugins/" + connectorClass +
"/config/validate";

            HttpHeaders headers = new HttpHeaders();
            headers.setContentType(MediaType.APPLICATION_JSON);

            String configJson = objectMapper.writeValueAsString(config);
            HttpEntity<String> entity = new HttpEntity<>(configJson, headers);

            ConnectorValidationResult result = restTemplate.exchange(
                url, HttpMethod.PUT, entity,
ConnectorValidationResult.class).getBody();

            return result != null ? result : new ConnectorValidationResult();

        } catch (Exception e) {
            log.error("Error validating connector config: {}", connectorClass, e);

            ConnectorValidationResult errorResult = new
ConnectorValidationResult();
            errorResult.setErrorCount(1);
            errorResult.setConfigs(Collections.singletonList(
                ConfigValidation.builder()

.definition(ConfigDefinition.builder().name("validation").build())
                    .value(ConfigValue.builder()
                        .errors(Collections.singletonList("Validation failed: " +
e.getMessage()))
                        .build())
                    .build()
            ));

            return errorResult;
        }
    }
}

/**
 * Connector configuration templates and builders
 */
@Component
@lombok.extern.slf4j.Slf4j
```

```
public class ConnectorConfigurationService {

    /**
     * Create JDBC Source Connector configuration
     */
    public Map<String, Object> createJdbcSourceConfig(JdbcSourceConfig config) {

        Map<String, Object> connectorConfig = new HashMap<>();

        // Connector class
        connectorConfig.put("connector.class",
"io.confluent.connect.jdbc.JdbcSourceConnector");

        // Connection configuration
        connectorConfig.put("connection.url", config.getConnectionUrl());
        connectorConfig.put("connection.user", config.getUsername());
        connectorConfig.put("connection.password", config.getPassword());

        // Mode and query configuration
        connectorConfig.put("mode", config.getMode()); // bulk, timestamp,
incrementing

        if ("timestamp".equals(config.getMode()) && config.getTimestampColumn() !=
null) {
            connectorConfig.put("timestamp.column.name",
config.getTimestampColumn());
        }

        if ("incrementing".equals(config.getMode()) &&
config.getIncrementingColumn() != null) {
            connectorConfig.put("incrementing.column.name",
config.getIncrementingColumn());
        }

        // Query configuration
        if (config.getQuery() != null) {
            connectorConfig.put("query", config.getQuery());
        } else if (config.getTableWhitelist() != null) {
            connectorConfig.put("table.whitelist", String.join(",",
config.getTableWhitelist()));
        }

        // Topic configuration
        if (config.getTopicPrefix() != null) {
            connectorConfig.put("topic.prefix", config.getTopicPrefix());
        }

        // Polling configuration
        connectorConfig.put("poll.interval.ms", config.getPollIntervalMs());
        connectorConfig.put("batch.max.rows", config.getBatchMaxRows());

        // Schema configuration
        connectorConfig.put("table.poll.interval.ms",
config.getTablePollIntervalMs());
```

```
        connectorConfig.put("schema.pattern", config.getSchemaPattern() != null ?
            config.getSchemaPattern() : "");

        log.info("Created JDBC Source Connector config: {}",
connectorConfig.keySet());

        return connectorConfig;
    }

    /**
     * Create JDBC Sink Connector configuration
     */
    public Map<String, Object> createJdbcSinkConfig(JdbcSinkConfig config) {

        Map<String, Object> connectorConfig = new HashMap<>();

        // Connector class
        connectorConfig.put("connector.class",
"io.confluent.connect.jdbc.JdbcSinkConnector");

        // Connection configuration
        connectorConfig.put("connection.url", config.getConnectionUrl());
        connectorConfig.put("connection.user", config.getUsername());
        connectorConfig.put("connection.password", config.getPassword());

        // Topic configuration
        connectorConfig.put("topics", String.join(",", config.getTopics()));

        // Table configuration
        if (config.getTableNameFormat() != null) {
            connectorConfig.put("table.name.format", config.getTableNameFormat());
        }

        // Insert mode
        connectorConfig.put("insert.mode", config.getInsertMode()); // insert,
upsert, update

        if ("upsert".equals(config.getInsertMode()) ||
"update".equals(config.getInsertMode())) {
            if (config.getPkFields() != null && !config.getPkFields().isEmpty()) {
                connectorConfig.put("pk.fields", String.join(",",
config.getPkFields()));
            }
            connectorConfig.put("pk.mode", config.getPkMode()); // none, kafka,
record_key, record_value
        }

        // Schema evolution
        connectorConfig.put("auto.create", config.isAutoCreate());
        connectorConfig.put("auto.evolve", config.isAutoEvolve());

        // Batch configuration
        connectorConfig.put("batch.size", config.getBatchSize());
        connectorConfig.put("max.retries", config.getMaxRetries());
```

```
        connectorConfig.put("retry.backoff.ms", config.getRetryBackoffMs());

        log.info("Created JDBC Sink Connector config: {}",
connectorConfig.keySet());

        return connectorConfig;
    }

    /**
     * Create File Source Connector configuration
     */
    public Map<String, Object> createFileSourceConfig(FileSourceConfig config) {

        Map<String, Object> connectorConfig = new HashMap<>();

        // Connector class
        connectorConfig.put("connector.class",
"org.apache.kafka.connect.file.FileStreamSourceConnector");

        // File configuration
        connectorConfig.put("file", config.getFilePath());
        connectorConfig.put("topic", config.getTopic());

        // Optional configurations
        if (config.getBatchSize() != null) {
            connectorConfig.put("batch.size", config.getBatchSize());
        }

        log.info("Created File Source Connector config for file: {}",
config.getFilePath());

        return connectorConfig;
    }

    /**
     * Create Elasticsearch Sink Connector configuration
     */
    public Map<String, Object>
createElasticsearchSinkConfig(ElasticsearchSinkConfig config) {

        Map<String, Object> connectorConfig = new HashMap<>();

        // Connector class
        connectorConfig.put("connector.class",
"io.confluent.connect.elasticsearch.ElasticsearchSinkConnector");

        // Connection configuration
        connectorConfig.put("connection.url", config.getConnectionUrl());
        if (config.getConnectionUsername() != null) {
            connectorConfig.put("connection.username",
config.getConnectionUsername());
            connectorConfig.put("connection.password",
config.getConnectionPassword());
        }
```

```
        // Topic configuration
        connectorConfig.put("topics", String.join(",", config.getTopics()));

        // Index configuration
        if (config.getTypeNameMapping() != null) {
            List<String> typeMappings =
config.getTypeNameMapping().entrySet().stream()
                .map(entry -> entry.getKey() + ":" + entry.getValue())
                .collect(Collectors.toList());
            connectorConfig.put("type.name", String.join(",", typeMappings));
        }

        // Batch configuration
        connectorConfig.put("batch.size", config.getBatchSize());
        connectorConfig.put("max.in.flight.requests",
config.getMaxInFlightRequests());
        connectorConfig.put("flush.timeout.ms", config.getFlushTimeoutMs());

        // Schema configuration
        connectorConfig.put("schema.ignore", config.isIgnoreSchema());
        connectorConfig.put("key.ignore", config.isIgnoreKey());

        log.info("Created Elasticsearch Sink Connector config: {}",
connectorConfig.keySet());

        return connectorConfig;
    }
}
```

This completes Part 2 covering Kafka Streams Integration and Kafka Connect Integration. The guide continues with Multi-tenancy setups, comparisons, and best practices in Part 3.