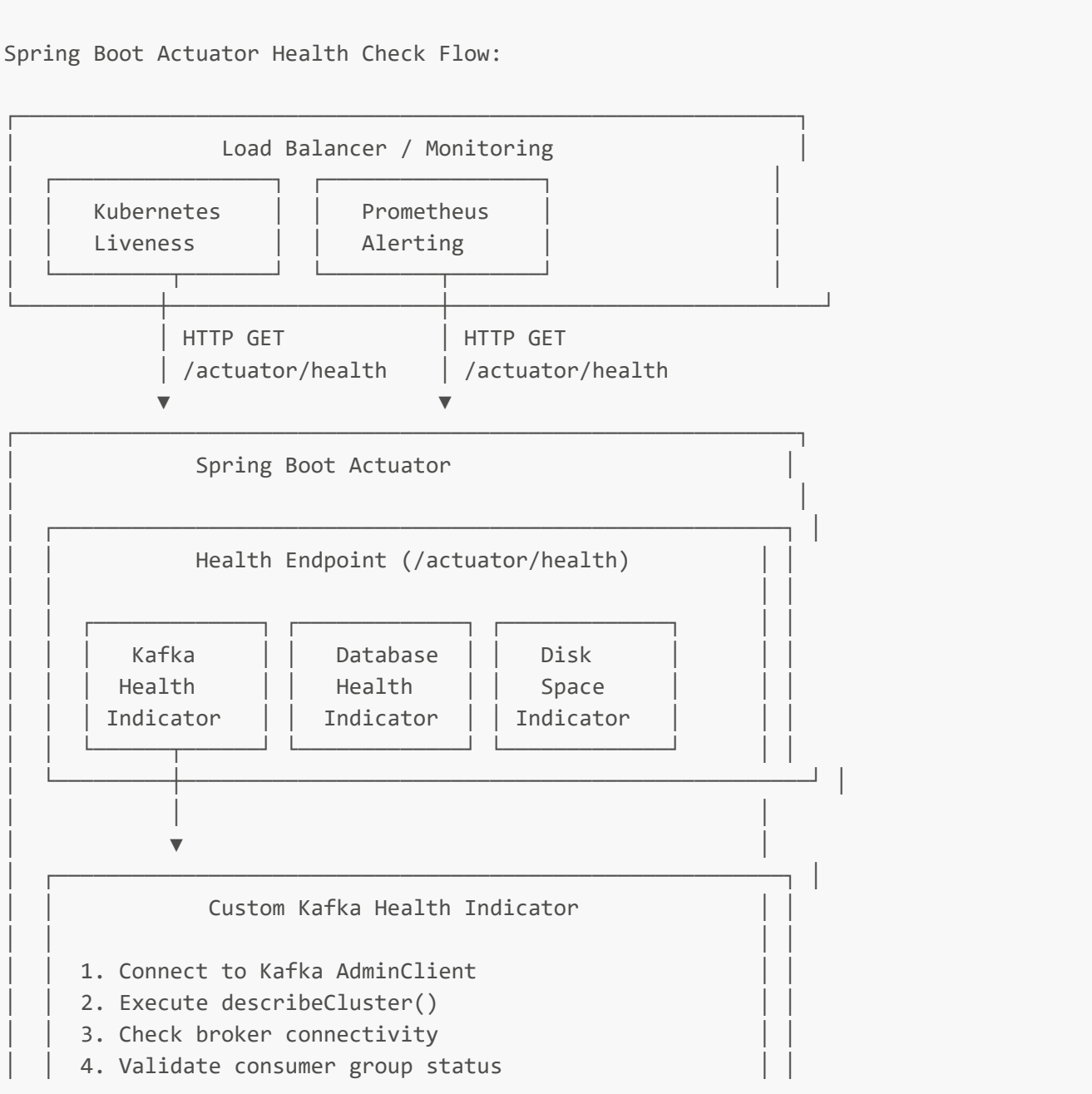
Spring Kafka Monitoring & Observability: Part 2 - Health Checks & Distributed Tracing

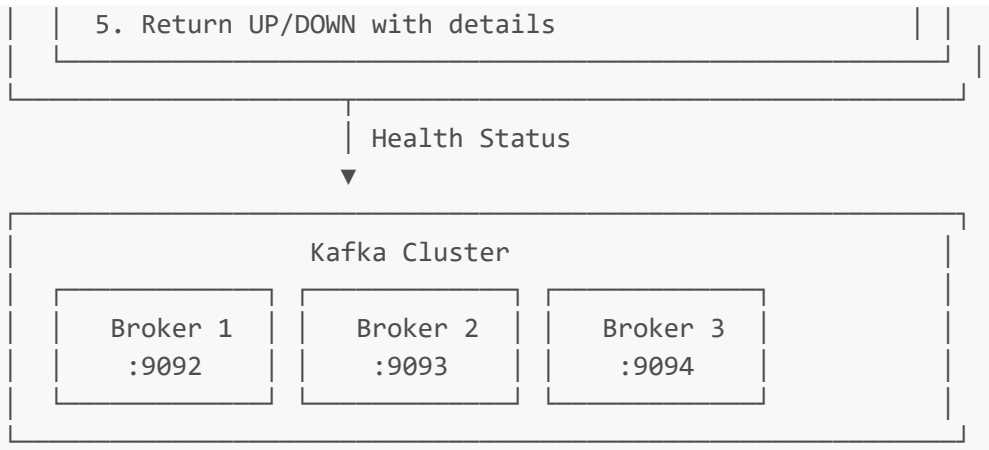
Continuation of the comprehensive Spring Kafka Monitoring & Observability guide covering health checks with Spring Boot Actuator and distributed tracing with Sleuth and OpenTelemetry.

🏥 Health Checks (Spring Boot Actuator)

Simple Explanation: Spring Boot Actuator provides health check endpoints that monitor the status of Kafka connectivity, broker availability, and overall system health. Health indicators automatically check if Kafka brokers are reachable and provide detailed health information that can be consumed by monitoring systems and load balancers.

Health Check Architecture:






```
Health Response Format:
{
  "status": "UP",
  "components": {
    "kafka": {
      "status": "UP",
      "details": {
        "clusterId": "kafka-cluster-1",
        "brokersCount": 3,
        "controllerId": 1,
        "responseTime": "45ms"
      }
    }
  }
}
```

Complete Health Check Implementation

```java
import org.springframework.boot.actuate.health.Health;
import org.springframework.boot.actuate.health.HealthIndicator;
import org.springframework.boot.actuate.health.Status;
import org.apache.kafka.clients.admin.AdminClient;
import org.apache.kafka.common.Node;

/**
 * Comprehensive Kafka health indicator implementation
 */
@Component
@lombok.extern.slf4j.Slf4j
public class KafkaHealthIndicator implements HealthIndicator {

    private final AdminClient adminClient;
    private final KafkaHealthProperties healthProperties;
    private final MeterRegistry meterRegistry;

    // Health check metrics
    private final Counter healthCheckSuccess;
    private final Counter healthCheckFailure;
```

```
    private final Timer healthCheckDuration;
    private final Gauge brokersCount;

    private volatile int lastKnownBrokerCount = 0;

    public KafkaHealthIndicator(AdminClient adminClient,
                                KafkaHealthProperties healthProperties,
                                MeterRegistry meterRegistry) {
        this.adminClient = adminClient;
        this.healthProperties = healthProperties;
        this.meterRegistry = meterRegistry;

        // Initialize health check metrics
        this.healthCheckSuccess = Counter.builder("kafka.health.check.success")
            .description("Number of successful Kafka health checks")
            .register(meterRegistry);

        this.healthCheckFailure = Counter.builder("kafka.health.check.failure")
            .description("Number of failed Kafka health checks")
            .register(meterRegistry);

        this.healthCheckDuration = Timer.builder("kafka.health.check.duration")
            .description("Duration of Kafka health checks")
            .register(meterRegistry);

        this.brokersCount = Gauge.builder("kafka.cluster.brokers.count")
            .description("Number of available Kafka brokers")
            .register(meterRegistry, this,
KafkaHealthIndicator::getLastKnownBrokerCount);
    }

    @Override
    public Health health() {

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

        try {
            log.debug("Performing Kafka health check");

            // Check cluster health
            ClusterHealth clusterHealth = checkClusterHealth();

            if (clusterHealth.isHealthy()) {

                healthCheckSuccess.increment();

                Health.Builder healthBuilder = Health.up()
                    .withDetail("clusterId", clusterHealth.getClusterId())
                    .withDetail("brokersCount", clusterHealth.getBrokerCount())
                    .withDetail("controllerId", clusterHealth.getControllerId())
                    .withDetail("responseTime", clusterHealth.getResponseTime())
                    .withDetail("lastChecked", Instant.now().toString());

                // Add broker details if configured
```

```
                if (healthProperties.isIncludeBrokerDetails()) {
                    healthBuilder.withDetail("brokers",
clusterHealth.getBrokerDetails());
                }

                // Add consumer group health if configured
                if (healthProperties.isCheckConsumerGroups()) {
                    ConsumerGroupHealth consumerGroupHealth =
checkConsumerGroupHealth();
                    healthBuilder.withDetail("consumerGroups",
consumerGroupHealth.getSummary());
                }

                // Update broker count
                lastKnownBrokerCount = clusterHealth.getBrokerCount();

                log.debug("Kafka health check passed: brokers={}, clusterId={}",
                    clusterHealth.getBrokerCount(), clusterHealth.getClusterId());

                return healthBuilder.build();

            } else {

                healthCheckFailure.increment();

                log.warn("Kafka health check failed: {}",
clusterHealth.getErrorMessage());

                return Health.down()
                    .withDetail("error", clusterHealth.getErrorMessage())
                    .withDetail("responseTime", clusterHealth.getResponseTime())
                    .withDetail("lastChecked", Instant.now().toString())
                    .build();
            }

        } catch (Exception e) {

            healthCheckFailure.increment();

            log.error("Kafka health check error", e);

            return Health.down()
                .withDetail("error", e.getMessage())
                .withDetail("errorType", e.getClass().getSimpleName())
                .withDetail("lastChecked", Instant.now().toString())
                .build();

        } finally {
            sample.stop(healthCheckDuration);
        }
    }

    private ClusterHealth checkClusterHealth() {
```

```
        try {
            long startTime = System.currentTimeMillis();

            // Get cluster description
            DescribeClusterResult clusterResult = adminClient.describeCluster();

            // Get cluster info with timeout
            String clusterId = clusterResult.clusterId().get(
                healthProperties.getTimeoutMs(), TimeUnit.MILLISECONDS);

            Collection<Node> nodes = clusterResult.nodes().get(
                healthProperties.getTimeoutMs(), TimeUnit.MILLISECONDS);

            Node controller = clusterResult.controller().get(
                healthProperties.getTimeoutMs(), TimeUnit.MILLISECONDS);

            long responseTime = System.currentTimeMillis() - startTime;

            // Validate broker count
            if (nodes.size() < healthProperties.getMinBrokers()) {
                return ClusterHealth.unhealthy(
                    String.format("Insufficient brokers: found %d, minimum
required %d",
                        nodes.size(), healthProperties.getMinBrokers()),
                    responseTime
                );
            }

            // Build broker details
            List<Map<String, Object>> brokerDetails = nodes.stream()
                .map(node -> Map.of(
                    "id", node.id(),
                    "host", node.host(),
                    "port", node.port(),
                    "rack", node.rack() != null ? node.rack() : "unknown"
                ))
                .collect(Collectors.toList());

            return ClusterHealth.healthy(
                clusterId,
                nodes.size(),
                controller.id(),
                responseTime,
                brokerDetails
            );

        } catch (TimeoutException e) {
            return ClusterHealth.unhealthy("Health check timeout",
healthProperties.getTimeoutMs());
        } catch (Exception e) {
            return ClusterHealth.unhealthy("Cluster check failed: " +
e.getMessage(), 0);
        }
    }
```

```
    private ConsumerGroupHealth checkConsumerGroupHealth() {

        try {
            // Get consumer groups
            ListConsumerGroupsResult groupsResult =
adminClient.listConsumerGroups();
            Collection<ConsumerGroupListing> groups = groupsResult.all().get(
                healthProperties.getTimeoutMs(), TimeUnit.MILLISECONDS);

            Map<String, Object> summary = new HashMap<>();
            summary.put("totalGroups", groups.size());
            summary.put("activeGroups", groups.stream()
                .filter(group -> group.state().isPresent() &&
                    group.state().get() == ConsumerGroupState.STABLE)
                .count());

            return new ConsumerGroupHealth(true, summary);

        } catch (Exception e) {
            log.warn("Failed to check consumer group health", e);
            Map<String, Object> summary = Map.of("error", e.getMessage());
            return new ConsumerGroupHealth(false, summary);
        }
    }

    public int getLastKnownBrokerCount() {
        return lastKnownBrokerCount;
    }

    // Inner classes for health data
    @lombok.Data
    @lombok.AllArgsConstructor
    private static class ClusterHealth {
        private boolean healthy;
        private String clusterId;
        private int brokerCount;
        private int controllerId;
        private long responseTime;
        private String errorMessage;
        private List<Map<String, Object>> brokerDetails;

        public static ClusterHealth healthy(String clusterId, int brokerCount, int
controllerId,
                                            long responseTime, List<Map<String,
Object>> brokerDetails) {
            return new ClusterHealth(true, clusterId, brokerCount, controllerId,
                responseTime, null, brokerDetails);
        }

        public static ClusterHealth unhealthy(String errorMessage, long
responseTime) {
            return new ClusterHealth(false, null, 0, -1, responseTime,
errorMessage, null);
```

```
        }
    }

    @lombok.Data
    @lombok.AllArgsConstructor
    private static class ConsumerGroupHealth {
        private boolean healthy;
        private Map<String, Object> summary;
    }
}

/**
 * Configuration properties for Kafka health checks
 */
@ConfigurationProperties(prefix = "management.health.kafka")
@lombok.Data
@Component
public class KafkaHealthProperties {

    private boolean enabled = true;
    private long timeoutMs = 5000; // 5 seconds
    private int minBrokers = 1;
    private boolean includeBrokerDetails = true;
    private boolean checkConsumerGroups = false;
    private Duration cacheTtl = Duration.ofSeconds(30);
}

/**
 * Advanced Kafka health indicator with caching and detailed checks
 */
@Component
@lombok.extern.slf4j.Slf4j
public class AdvancedKafkaHealthIndicator implements HealthIndicator {

    private final AdminClient adminClient;
    private final KafkaHealthProperties properties;
    private final MeterRegistry meterRegistry;

    // Cached health result
    private volatile CachedHealthResult cachedResult;

    // Advanced metrics
    private final Timer brokerConnectionTime;
    private final Counter topicMetadataErrors;
    private final Gauge consumerGroupLag;

    public AdvancedKafkaHealthIndicator(AdminClient adminClient,
                                        KafkaHealthProperties properties,
                                        MeterRegistry meterRegistry) {
        this.adminClient = adminClient;
        this.properties = properties;
        this.meterRegistry = meterRegistry;

        // Initialize advanced metrics
```

```
        this.brokerConnectionTime =
Timer.builder("kafka.health.broker.connection.time")
            .description("Time to connect to each Kafka broker")
            .register(meterRegistry);

        this.topicMetadataErrors =
Counter.builder("kafka.health.topic.metadata.errors")
            .description("Number of topic metadata retrieval errors")
            .register(meterRegistry);

        this.consumerGroupLag =
Gauge.builder("kafka.health.consumer.group.max.lag")
            .description("Maximum consumer group lag across all groups")
            .register(meterRegistry, this,
AdvancedKafkaHealthIndicator::calculateMaxConsumerLag);
    }

    @Override
    public Health health() {

        // Check if cached result is still valid
        if (cachedResult != null && cachedResult.isValid()) {
            log.debug("Returning cached Kafka health result");
            return cachedResult.getHealth();
        }

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

        try {
            Health health = performDetailedHealthCheck();

            // Cache the result
            cachedResult = new CachedHealthResult(health, Instant.now(),
properties.getCacheTtl());

            return health;

        } finally {
            sample.stop(Timer.builder("kafka.health.check.total.duration")
                .register(meterRegistry));
        }
    }

    private Health performDetailedHealthCheck() {

        Health.Builder healthBuilder = Health.up();
        boolean overallHealthy = true;
        Map<String, Object> details = new HashMap<>();

        try {
            // 1. Check cluster connectivity
            ClusterConnectivityResult connectivity = checkClusterConnectivity();
            details.put("cluster", connectivity.toMap());
            if (!connectivity.isHealthy()) overallHealthy = false;
```

```
            // 2. Check broker individual health
            BrokerHealthResult brokerHealth = checkIndividualBrokers();
            details.put("brokers", brokerHealth.toMap());
            if (!brokerHealth.isHealthy()) overallHealthy = false;

            // 3. Check topic metadata accessibility
            TopicMetadataResult topicMetadata = checkTopicMetadata();
            details.put("topics", topicMetadata.toMap());
            if (!topicMetadata.isHealthy()) overallHealthy = false;

            // 4. Check consumer groups (if enabled)
            if (properties.isCheckConsumerGroups()) {
                ConsumerGroupResult consumerGroups = checkConsumerGroups();
                details.put("consumerGroups", consumerGroups.toMap());
                if (!consumerGroups.isHealthy()) overallHealthy = false;
            }

            // 5. Performance checks
            PerformanceResult performance = checkPerformance();
            details.put("performance", performance.toMap());

            details.put("lastChecked", Instant.now().toString());
            details.put("healthCheckVersion", "2.0");

            if (overallHealthy) {
                return healthBuilder.withDetails(details).build();
            } else {
                return Health.down().withDetails(details).build();
            }

        } catch (Exception e) {
            log.error("Advanced Kafka health check failed", e);

            return Health.down()
                .withDetail("error", e.getMessage())
                .withDetail("errorType", e.getClass().getSimpleName())
                .withDetail("lastChecked", Instant.now().toString())
                .build();
        }
    }

    private ClusterConnectivityResult checkClusterConnectivity() {

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

        try {
            DescribeClusterResult result = adminClient.describeCluster();

            String clusterId = result.clusterId().get(properties.getTimeoutMs(),
TimeUnit.MILLISECONDS);
            Collection<Node> nodes = result.nodes().get(properties.getTimeoutMs(),
TimeUnit.MILLISECONDS);
            Node controller = result.controller().get(properties.getTimeoutMs(),
```

```
TimeUnit.MILLISECONDS);

            sample.stop(Timer.builder("kafka.health.cluster.connectivity.time")
                .register(meterRegistry));

            return ClusterConnectivityResult.healthy(clusterId, nodes.size(),
controller.id());

        } catch (Exception e) {
            sample.stop(Timer.builder("kafka.health.cluster.connectivity.time")
                .tag("result", "error")
                .register(meterRegistry));

            return ClusterConnectivityResult.unhealthy(e.getMessage());
        }
    }

    private BrokerHealthResult checkIndividualBrokers() {

        try {
            DescribeClusterResult clusterResult = adminClient.describeCluster();
            Collection<Node> nodes =
clusterResult.nodes().get(properties.getTimeoutMs(), TimeUnit.MILLISECONDS);

            List<Map<String, Object>> brokerStatuses = new ArrayList<>();
            int healthyBrokers = 0;

            for (Node node : nodes) {
                Timer.Sample brokerSample = brokerConnectionTime.start();

                try {
                    // Test connection to individual broker
                    boolean brokerHealthy = testBrokerConnectivity(node);

                    brokerSample.stop();

                    Map<String, Object> brokerStatus = Map.of(
                        "id", node.id(),
                        "host", node.host(),
                        "port", node.port(),
                        "rack", node.rack() != null ? node.rack() : "unknown",
                        "status", brokerHealthy ? "UP" : "DOWN"
                    );

                    brokerStatuses.add(brokerStatus);

                    if (brokerHealthy) healthyBrokers++;

                } catch (Exception e) {
                    brokerSample.stop(Tags.of("result", "error"));

                    Map<String, Object> brokerStatus = Map.of(
                        "id", node.id(),
                        "host", node.host(),
```

```
                        "port", node.port(),
                        "status", "ERROR",
                        "error", e.getMessage()
                    );

                    brokerStatuses.add(brokerStatus);
                }
            }

            boolean overallHealthy = healthyBrokers >= properties.getMinBrokers();

            return new BrokerHealthResult(overallHealthy, healthyBrokers,
nodes.size(), brokerStatuses);

        } catch (Exception e) {
            return new BrokerHealthResult(false, 0, 0,
                Collections.singletonList(Map.of("error", e.getMessage())));
        }
    }

    private TopicMetadataResult checkTopicMetadata() {

        try {
            ListTopicsResult topicsResult = adminClient.listTopics();
            Set<String> topicNames =
topicsResult.names().get(properties.getTimeoutMs(), TimeUnit.MILLISECONDS);

            if (topicNames.isEmpty()) {
                return new TopicMetadataResult(true, 0, "No topics found");
            }

            // Test metadata access for a sample of topics
            int sampleSize = Math.min(topicNames.size(), 5);
            List<String> sampleTopics = topicNames.stream()
                .limit(sampleSize)
                .collect(Collectors.toList());

            DescribeTopicsResult describeResult =
adminClient.describeTopics(sampleTopics);
            Map<String, TopicDescription> descriptions =
                describeResult.all().get(properties.getTimeoutMs(),
TimeUnit.MILLISECONDS);

            return new TopicMetadataResult(true, topicNames.size(),
                String.format("Successfully accessed metadata for %d topics",
descriptions.size()));

        } catch (Exception e) {
            topicMetadataErrors.increment();
            return new TopicMetadataResult(false, -1, "Topic metadata access
failed: " + e.getMessage());
        }
    }
```

```
    private ConsumerGroupResult checkConsumerGroups() {

        try {
            ListConsumerGroupsResult groupsResult =
adminClient.listConsumerGroups();
            Collection<ConsumerGroupListing> groups =
                groupsResult.all().get(properties.getTimeoutMs(),
TimeUnit.MILLISECONDS);

            long stableGroups = groups.stream()
                .filter(group -> group.state().isPresent() &&
                    group.state().get() == ConsumerGroupState.STABLE)
                .count();

            return new ConsumerGroupResult(true, groups.size(), stableGroups);

        } catch (Exception e) {
            return new ConsumerGroupResult(false, -1, -1, "Consumer group check
failed: " + e.getMessage());
        }
    }

    private PerformanceResult checkPerformance() {

        long startTime = System.currentTimeMillis();

        try {
            // Perform a lightweight operation to measure responsiveness
            adminClient.listTopics().names().get(1000, TimeUnit.MILLISECONDS);

            long responseTime = System.currentTimeMillis() - startTime;

            return new PerformanceResult(true, responseTime,
                responseTime < 1000 ? "GOOD" : "SLOW");

        } catch (Exception e) {
            long responseTime = System.currentTimeMillis() - startTime;
            return new PerformanceResult(false, responseTime, "FAILED: " +
e.getMessage());
        }
    }

    private boolean testBrokerConnectivity(Node node) {
        // Simplified broker connectivity test
        // In production, you might want to test actual socket connectivity
        return true;
    }

    private double calculateMaxConsumerLag(AdvancedKafkaHealthIndicator indicator)
{
        // Simplified lag calculation
        // In production, implement actual consumer lag calculation
        return 0.0;
    }
```

```
    // Result classes
    @lombok.Data
    @lombok.AllArgsConstructor
    private static class ClusterConnectivityResult {
        private boolean healthy;
        private String clusterId;
        private int brokerCount;
        private int controllerId;
        private String errorMessage;

        public static ClusterConnectivityResult healthy(String clusterId, int
brokerCount, int controllerId) {
            return new ClusterConnectivityResult(true, clusterId, brokerCount,
controllerId, null);
        }

        public static ClusterConnectivityResult unhealthy(String errorMessage) {
            return new ClusterConnectivityResult(false, null, 0, -1,
errorMessage);
        }

        public Map<String, Object> toMap() {
            Map<String, Object> map = new HashMap<>();
            map.put("healthy", healthy);
            if (healthy) {
                map.put("clusterId", clusterId);
                map.put("brokerCount", brokerCount);
                map.put("controllerId", controllerId);
            } else {
                map.put("error", errorMessage);
            }
            return map;
        }
    }

    @lombok.Data
    @lombok.AllArgsConstructor
    private static class BrokerHealthResult {
        private boolean healthy;
        private int healthyBrokers;
        private int totalBrokers;
        private List<Map<String, Object>> brokerStatuses;

        public Map<String, Object> toMap() {
            return Map.of(
                "healthy", healthy,
                "healthyBrokers", healthyBrokers,
                "totalBrokers", totalBrokers,
                "brokers", brokerStatuses
            );
        }
    }
```

```
    @lombok.Data
    @lombok.AllArgsConstructor
    private static class TopicMetadataResult {
        private boolean healthy;
        private int topicCount;
        private String message;

        public Map<String, Object> toMap() {
            return Map.of(
                "healthy", healthy,
                "topicCount", topicCount,
                "message", message
            );
        }
    }

    @lombok.Data
    @lombok.AllArgsConstructor
    private static class ConsumerGroupResult {
        private boolean healthy;
        private long totalGroups;
        private long stableGroups;
        private String errorMessage;

        public ConsumerGroupResult(boolean healthy, long totalGroups, long
stableGroups) {
            this(healthy, totalGroups, stableGroups, null);
        }

        public Map<String, Object> toMap() {
            Map<String, Object> map = new HashMap<>();
            map.put("healthy", healthy);
            map.put("totalGroups", totalGroups);
            map.put("stableGroups", stableGroups);
            if (errorMessage != null) {
                map.put("error", errorMessage);
            }
            return map;
        }
    }

    @lombok.Data
    @lombok.AllArgsConstructor
    private static class PerformanceResult {
        private boolean healthy;
        private long responseTimeMs;
        private String status;

        public Map<String, Object> toMap() {
            return Map.of(
                "healthy", healthy,
                "responseTimeMs", responseTimeMs,
                "status", status
            );
```

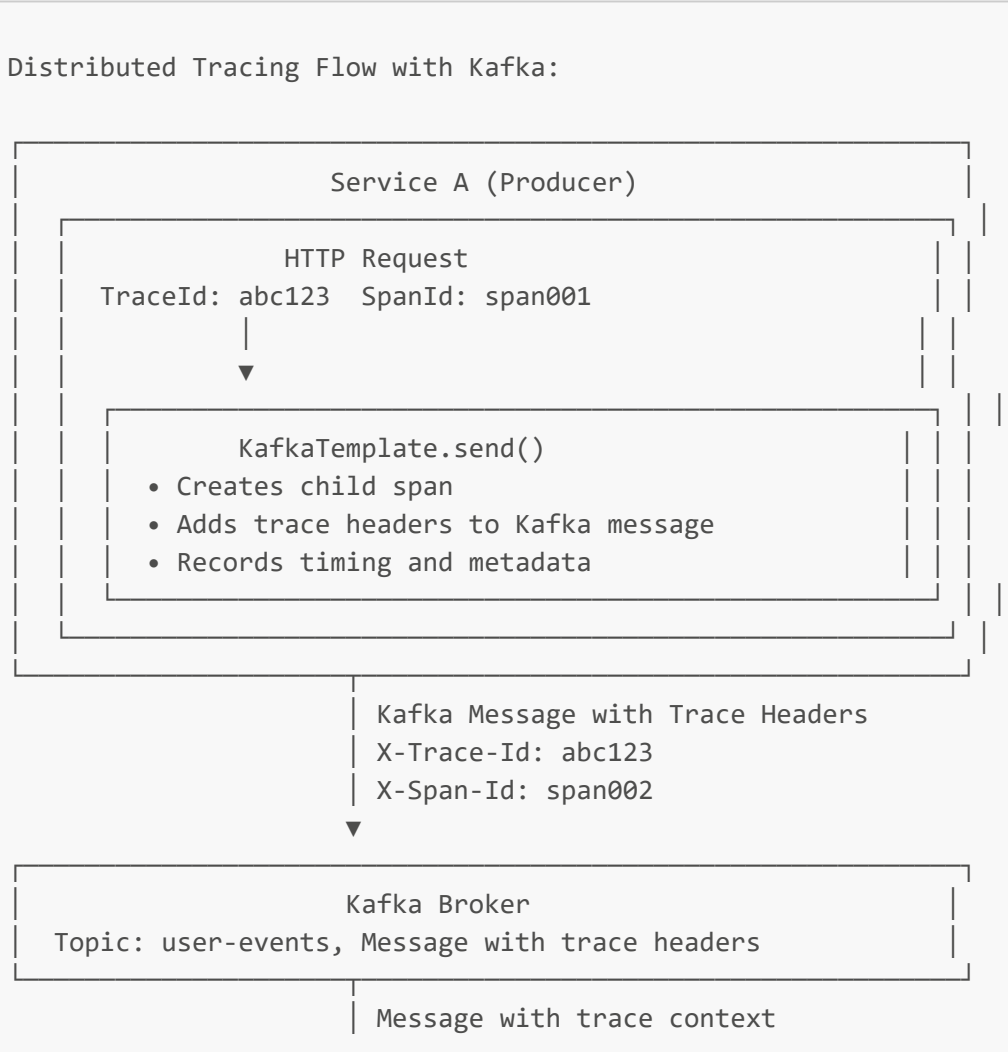
```
        }
    }

    @lombok.Data
    @lombok.AllArgsConstructor
    private static class CachedHealthResult {
        private Health health;
        private Instant timestamp;
        private Duration ttl;

        public boolean isValid() {
            return Instant.now().isBefore(timestamp.plus(ttl));
        }
    }
}

/**
 * Health check configuration
 */
@Configuration
@lombok.extern.slf4j.Slf4j
public class KafkaHealthConfiguration {

    /**
     * Configure health check properties
     */
    @Bean
    @ConfigurationProperties(prefix = "management.health.kafka")
    public KafkaHealthProperties kafkaHealthProperties() {
        return new KafkaHealthProperties();
    }

    /**
     * Conditional health indicator registration
     */
    @Bean
    @ConditionalOnProperty(prefix = "management.health.kafka", name = "enabled",
havingValue = "true", matchIfMissing = true)
    public KafkaHealthIndicator kafkaHealthIndicator(AdminClient adminClient,
                                                     KafkaHealthProperties
properties,
                                                     MeterRegistry meterRegistry) {

        log.info("Registering Kafka health indicator with properties: {}",
properties);

        return new KafkaHealthIndicator(adminClient, properties, meterRegistry);
    }

    /**
     * Advanced health indicator (optional)
     */
    @Bean
    @ConditionalOnProperty(prefix = "management.health.kafka", name = "advanced",
```

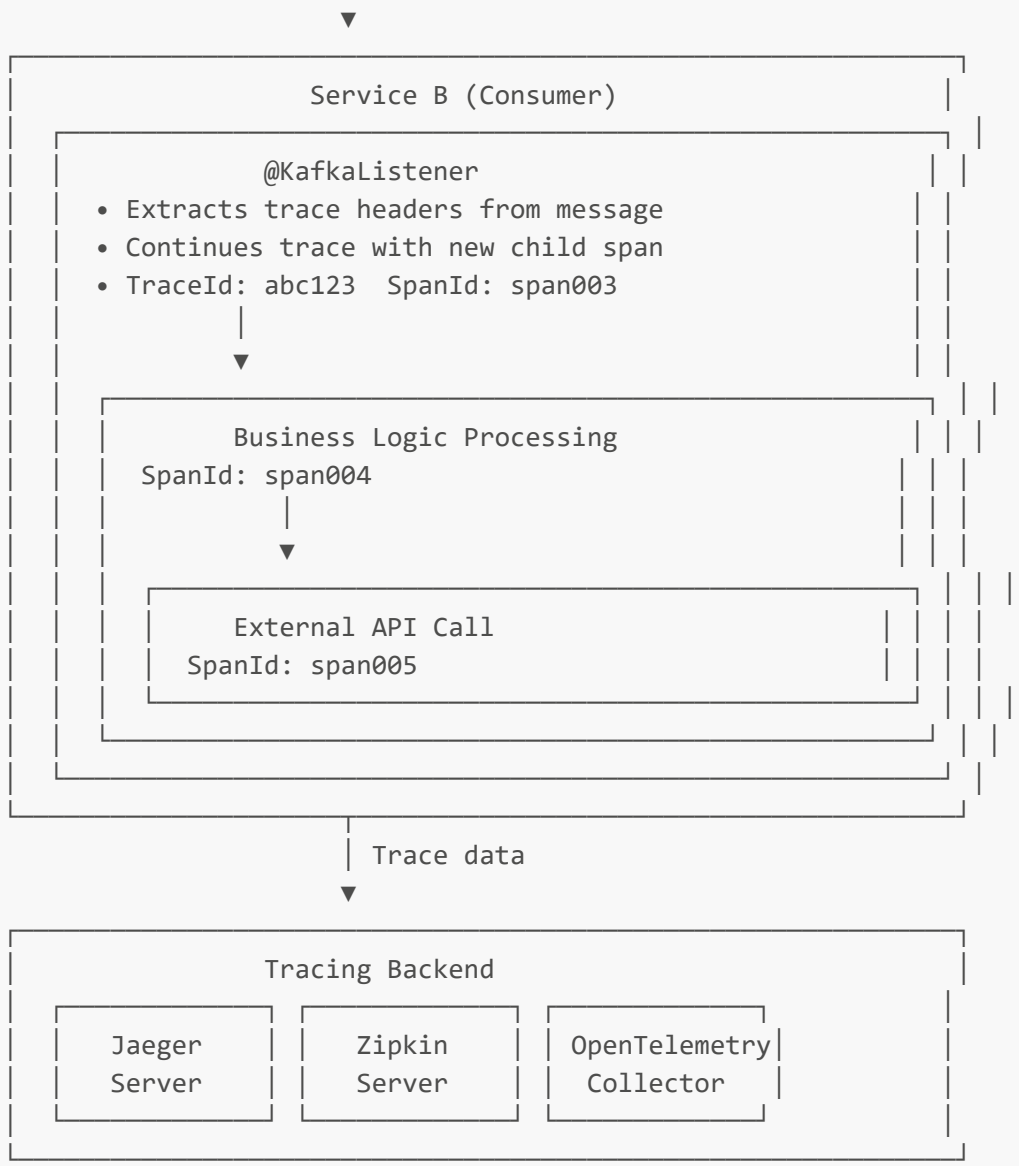
```
havingValue = "true")
    public AdvancedKafkaHealthIndicator advancedKafkaHealthIndicator(AdminClient
adminClient,

KafkaHealthProperties properties,
                                                                    MeterRegistry
meterRegistry) {

        log.info("Registering advanced Kafka health indicator");

        return new AdvancedKafkaHealthIndicator(adminClient, properties,
meterRegistry);
    }
}
```

🌐 Distributed Tracing (Sleuth, OpenTelemetry)

Simple Explanation: Distributed tracing tracks requests as they flow through multiple services and systems, creating a trace that shows the complete journey of a message from producer to consumer. Spring Cloud Sleuth and OpenTelemetry provide automatic instrumentation for Kafka operations, capturing timing, spans, and correlation IDs that help debug performance issues and understand system behavior.

Distributed Tracing Architecture:






```
Trace Visualization:
TraceId: abc123
├── span001 (HTTP Request) [Service A] 45ms
│   └── span002 (Kafka Send) [Service A] 12ms
├── span003 (Kafka Receive) [Service B] 2ms
│   ├── span004 (Business Logic) [Service B] 25ms
│   └── span005 (External API) [Service B] 150ms
Total Duration: 234ms
```

Complete Spring Cloud Sleuth Configuration

```
import org.springframework.cloud.sleuth.Tracer;
import org.springframework.cloud.sleuth.Span;
import org.springframework.cloud.sleuth.TraceContext;

/**
 * Spring Cloud Sleuth configuration for Kafka tracing
 */
@Configuration
```

```
@ConditionalOnClass(Tracer.class)
@lombok.extern.slf4j.Slf4j
public class KafkaSleuthConfiguration {

    /**
     * Kafka template with Sleuth observability
     */
    @Bean
    public KafkaTemplate<String, Object>
sleuthKafkaTemplate(ProducerFactory<String, Object> producerFactory) {

        KafkaTemplate<String, Object> template = new KafkaTemplate<>
(producerFactory);

        // Enable observations for Sleuth integration
        template.setObservationEnabled(true);

        // Set custom observation convention for better tracing
        template.setObservationConvention(new
SleuthKafkaTemplateObservationConvention());

        log.info("Configured KafkaTemplate with Sleuth tracing");

        return template;
    }

    /**
     * Listener container factory with Sleuth observability
     */
    @Bean
    public ConcurrentKafkaListenerContainerFactory<String, Object>
sleuthKafkaListenerContainerFactory(
            ConsumerFactory<String, Object> consumerFactory) {

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

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

        // Enable observations for Sleuth integration
        factory.getContainerProperties().setObservationEnabled(true);

        // Set custom observation convention
        factory.setContainerCustomizer(container -> {
            container.getContainerProperties().setObservationConvention(
                new SleuthKafkaListenerObservationConvention());
        });

        log.info("Configured KafkaListenerContainerFactory with Sleuth tracing");

        return factory;
    }
```

```
    /**
     * Custom observation convention for producer tracing
     */
    public static class SleuthKafkaTemplateObservationConvention implements
KafkaTemplateObservationConvention {

        @Override
        public String getName() {
            return "kafka.producer";
        }

        @Override
        public String getContextualName(KafkaRecordSenderContext context) {
            return "kafka.producer " + context.getDestination();
        }

        @Override
        public KeyValues getLowCardinalityKeyValues(KafkaRecordSenderContext
context) {
            return KeyValues.of(
                "messaging.system", "kafka",
                "messaging.destination", context.getDestination(),
                "messaging.operation", "send",
                "messaging.destination_kind", "topic"
            );
        }

        @Override
        public KeyValues getHighCardinalityKeyValues(KafkaRecordSenderContext
context) {
            return KeyValues.of(
                "messaging.message_id", String.valueOf(context.getRecord().key()),
                "kafka.partition",
String.valueOf(context.getRecord().partition()),
                "messaging.kafka.message_key",
String.valueOf(context.getRecord().key())
            );
        }
    }

    /**
     * Custom observation convention for consumer tracing
     */
    public static class SleuthKafkaListenerObservationConvention implements
KafkaListenerObservationConvention {

        @Override
        public String getName() {
            return "kafka.consumer";
        }

        @Override
        public String getContextualName(KafkaRecordReceiverContext context) {
            return "kafka.consumer " + context.getSource();
```

```
        }

        @Override
        public KeyValues getLowCardinalityKeyValues(KafkaRecordReceiverContext
context) {
            return KeyValues.of(
                "messaging.system", "kafka",
                "messaging.destination", context.getSource(),
                "messaging.operation", "receive",
                "messaging.destination_kind", "topic",
                "kafka.consumer.group", context.getGroupId() != null ?
context.getGroupId() : "unknown"
            );
        }

        @Override
        public KeyValues getHighCardinalityKeyValues(KafkaRecordReceiverContext
context) {
            return KeyValues.of(
                "messaging.message_id", String.valueOf(context.getRecord().key()),
                "kafka.partition",
String.valueOf(context.getRecord().partition()),
                "kafka.offset", String.valueOf(context.getRecord().offset()),
                "messaging.kafka.message_key",
String.valueOf(context.getRecord().key())
            );
        }
    }
}

/**
 * Instrumented service with manual tracing capabilities
 */
@Service
@lombok.extern.slf4j.Slf4j
public class TracedKafkaService {

    @Autowired
    private KafkaTemplate<String, Object> kafkaTemplate;

    @Autowired
    private Tracer tracer;

    /**
     * Send message with custom span creation
     */
    public void sendTracedMessage(String topic, String key, Object message) {

        // Create custom span for business logic
        Span businessSpan = tracer.nextSpan()
            .name("business.message.preparation")
            .tag("business.operation", "message.send")
            .tag("topic", topic)
            .tag("message.type", message.getClass().getSimpleName())
```

```
            .start();

        try (Tracer.SpanInScope ws = tracer.withSpanInScope(businessSpan)) {

            log.info("Preparing traced message: topic={}, key={}", topic, key);

            // Simulate business logic processing
            processBusinessLogic(message);

            // Add business context to span
            businessSpan.tag("message.processed", "true");
            businessSpan.tag("processing.duration", "25ms");

            // Send message (automatically traced by Sleuth)
            ListenableFuture<SendResult<String, Object>> future =
kafkaTemplate.send(topic, key, message);

            // Add callback with tracing context
            future.addCallback(
                result -> handleSendSuccess(result, topic, key),
                failure -> handleSendFailure(failure, topic, key)
            );

            businessSpan.tag("message.sent", "true");

        } catch (Exception e) {
            businessSpan.tag("error", e.getMessage());
            throw e;
        } finally {
            businessSpan.end();
        }
    }

    /**
     * Traced message consumer
     */
    @KafkaListener(
        topics = "traced-events",
        groupId = "traced-consumer-group",
        containerFactory = "sleuthKafkaListenerContainerFactory"
    )
    public void consumeTracedMessage(@Payload TracedEvent event,
                                     @Header(KafkaHeaders.RECEIVED_TOPIC) String
topic,
                                     @Header(KafkaHeaders.RECEIVED_PARTITION) int
partition,
                                     @Header(KafkaHeaders.OFFSET) long offset,
                                     Acknowledgment ack) {

        // Get current trace context (automatically provided by Sleuth)
        Span currentSpan = tracer.currentSpan();
        if (currentSpan != null) {
            currentSpan.tag("kafka.topic", topic);
            currentSpan.tag("kafka.partition", String.valueOf(partition));
```

```
            currentSpan.tag("kafka.offset", String.valueOf(offset));
            currentSpan.tag("event.id", event.getEventId());
        }

        // Create custom span for business processing
        Span processSpan = tracer.nextSpan()
            .name("business.event.processing")
            .tag("event.type", event.getEventType())
            .tag("event.id", event.getEventId())
            .start();

        try (Tracer.SpanInScope ws = tracer.withSpanInScope(processSpan)) {

            log.info("Processing traced event: eventId={}, type={}, topic={},
offset={}",
                event.getEventId(), event.getEventType(), topic, offset);

            // Process the event
            processTracedEvent(event);

            // Add processing results to span
            processSpan.tag("processing.successful", "true");
            processSpan.tag("processing.duration", "45ms");

            // Manual acknowledgment
            ack.acknowledge();

            log.info("Traced event processed successfully: eventId={}",
event.getEventId());

        } catch (Exception e) {

            processSpan.tag("error", e.getMessage());
            processSpan.tag("processing.successful", "false");

            log.error("Error processing traced event: eventId={}",
event.getEventId(), e);
            throw e;

        } finally {
            processSpan.end();
        }
    }

    /**
     * Chain of traced operations
     */
    public void chainedTracedOperations(String userId, String action) {

        Span chainSpan = tracer.nextSpan()
            .name("business.chained.operations")
            .tag("user.id", userId)
            .tag("action", action)
            .start();
```

```
        try (Tracer.SpanInScope ws = tracer.withSpanInScope(chainSpan)) {

            // Step 1: Create user event
            TracedEvent userEvent = createUserEvent(userId, action);
            sendTracedMessage("user-events", userId, userEvent);

            // Step 2: Create audit event
            TracedEvent auditEvent = createAuditEvent(userId, action);
            sendTracedMessage("audit-events", userId, auditEvent);

            // Step 3: External API call (also traced)
            callExternalService(userId, action);

            chainSpan.tag("operations.completed", "3");

        } catch (Exception e) {
            chainSpan.tag("error", e.getMessage());
            throw e;
        } finally {
            chainSpan.end();
        }
    }

    // Helper methods
    private void processBusinessLogic(Object message) {
        // Simulate processing
        try {
            Thread.sleep(25);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }

    private void processTracedEvent(TracedEvent event) {
        // Simulate event processing
        try {
            Thread.sleep(45);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }

    private TracedEvent createUserEvent(String userId, String action) {
        return new TracedEvent(UUID.randomUUID().toString(), "USER_ACTION",
userId, action, Instant.now());
    }

    private TracedEvent createAuditEvent(String userId, String action) {
        return new TracedEvent(UUID.randomUUID().toString(), "AUDIT_LOG", userId,
action, Instant.now());
    }

    @NewSpan("external.service.call")
```

```
    private void callExternalService(String userId, String action) {

        // Simulate external service call
        try {
            Thread.sleep(100);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }

        // Add span tags
        if (tracer.currentSpan() != null) {
            tracer.currentSpan().tag("external.service", "user-service");
            tracer.currentSpan().tag("user.id", userId);
        }
    }

    private void handleSendSuccess(SendResult<String, Object> result, String
topic, String key) {

        log.debug("Traced message sent successfully: topic={}, key={}, offset={}",
            topic, key, result.getRecordMetadata().offset());

        // Add span information if available
        if (tracer.currentSpan() != null) {
            tracer.currentSpan().tag("kafka.send.success", "true");
            tracer.currentSpan().tag("kafka.offset",
String.valueOf(result.getRecordMetadata().offset()));
        }
    }

    private void handleSendFailure(Throwable failure, String topic, String key) {

        log.error("Traced message send failed: topic={}, key={}", topic, key,
failure);

        // Add error information to span
        if (tracer.currentSpan() != null) {
            tracer.currentSpan().tag("kafka.send.success", "false");
            tracer.currentSpan().tag("error", failure.getMessage());
        }
    }
}

// Supporting data classes
@lombok.Data
@lombok.AllArgsConstructor
@lombok.NoArgsConstructor
class TracedEvent {
    private String eventId;
    private String eventType;
    private String userId;
    private String action;
    private Instant timestamp;
}
```

Complete OpenTelemetry Configuration

```
import io.opentelemetry.api.OpenTelemetry;
import io.opentelemetry.api.trace.Tracer;
import io.opentelemetry.api.trace.Span;
import io.opentelemetry.context.Scope;

/**
 * OpenTelemetry configuration for Kafka tracing
 */
@Configuration
@ConditionalOnClass(OpenTelemetry.class)
@lombok.extern.slf4j.Slf4j
public class KafkaOpenTelemetryConfiguration {

    @Autowired
    private OpenTelemetry openTelemetry;

    private final Tracer tracer = openTelemetry.getTracer("kafka-service",
"1.0.0");

    /**
     * Kafka template with OpenTelemetry observability
     */
    @Bean
    public KafkaTemplate<String, Object> otelKafkaTemplate(ProducerFactory<String,
Object> producerFactory) {

        KafkaTemplate<String, Object> template = new KafkaTemplate<>
(producerFactory);

        // Enable observations for OpenTelemetry integration
        template.setObservationEnabled(true);

        // Set custom observation convention
        template.setObservationConvention(new
OpenTelemetryKafkaTemplateObservationConvention());

        // Add interceptors for additional tracing
        template.setProducerInterceptors(Arrays.asList(new
OpenTelemetryProducerInterceptor()));

        log.info("Configured KafkaTemplate with OpenTelemetry tracing");

        return template;
    }

    /**
     * Consumer factory with OpenTelemetry tracing
     */
```

```
    @Bean
    public ConcurrentKafkaListenerContainerFactory<String, Object>
otelKafkaListenerContainerFactory(
            ConsumerFactory<String, Object> consumerFactory) {

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

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

        // Enable observations for OpenTelemetry
        factory.getContainerProperties().setObservationEnabled(true);

        // Set custom observation convention
        factory.setContainerCustomizer(container -> {
            container.getContainerProperties().setObservationConvention(
                new OpenTelemetryKafkaListenerObservationConvention());
        });

        // Add consumer interceptors for tracing
        factory.setConsumerInterceptors(Arrays.asList(new
OpenTelemetryConsumerInterceptor()));

        log.info("Configured KafkaListenerContainerFactory with OpenTelemetry
tracing");

        return factory;
    }

    /**
     * OpenTelemetry observation convention for producer
     */
    public static class OpenTelemetryKafkaTemplateObservationConvention implements
KafkaTemplateObservationConvention {

        @Override
        public String getName() {
            return "kafka.producer.send";
        }

        @Override
        public String getContextualName(KafkaRecordSenderContext context) {
            return "kafka send " + context.getDestination();
        }

        @Override
        public KeyValues getLowCardinalityKeyValues(KafkaRecordSenderContext
context) {
            return KeyValues.of(
                // OpenTelemetry semantic conventions
                "messaging.system", "kafka",
                "messaging.destination.name", context.getDestination(),
                "messaging.operation", "publish",
```

```
                "messaging.destination.kind", "topic"
            );
        }

        @Override
        public KeyValues getHighCardinalityKeyValues(KafkaRecordSenderContext
context) {
            return KeyValues.of(
                "messaging.message.id", String.valueOf(context.getRecord().key()),
                "messaging.kafka.destination.partition",
String.valueOf(context.getRecord().partition()),
                "messaging.kafka.message.key",
String.valueOf(context.getRecord().key()),
                "messaging.message.payload_size_bytes",
String.valueOf(estimateMessageSize(context.getRecord()))
            );
        }

        private long estimateMessageSize(ProducerRecord<String, Object> record) {
            long size = 0;
            if (record.key() != null) size += record.key().getBytes().length;
            if (record.value() != null) size +=
record.value().toString().getBytes().length;
            return size;
        }
    }

    /**
     * OpenTelemetry observation convention for consumer
     */
    public static class OpenTelemetryKafkaListenerObservationConvention implements
KafkaListenerObservationConvention {

        @Override
        public String getName() {
            return "kafka.consumer.receive";
        }

        @Override
        public String getContextualName(KafkaRecordReceiverContext context) {
            return "kafka receive " + context.getSource();
        }

        @Override
        public KeyValues getLowCardinalityKeyValues(KafkaRecordReceiverContext
context) {
            return KeyValues.of(
                "messaging.system", "kafka",
                "messaging.destination.name", context.getSource(),
                "messaging.operation", "receive",
                "messaging.destination.kind", "topic",
                "messaging.kafka.consumer.group", context.getGroupId() != null ?
context.getGroupId() : "unknown"
            );
```

```
        }

        @Override
        public KeyValues getHighCardinalityKeyValues(KafkaRecordReceiverContext
context) {
            return KeyValues.of(
                "messaging.message.id", String.valueOf(context.getRecord().key()),
                "messaging.kafka.source.partition",
String.valueOf(context.getRecord().partition()),
                "messaging.kafka.message.offset",
String.valueOf(context.getRecord().offset()),
                "messaging.kafka.message.key",
String.valueOf(context.getRecord().key()),
                "messaging.message.payload_size_bytes",
String.valueOf(estimateMessageSize(context.getRecord()))
            );
        }

        private long estimateMessageSize(ConsumerRecord<String, Object> record) {
            long size = 0;
            if (record.key() != null) size += record.key().getBytes().length;
            if (record.value() != null) size +=
record.value().toString().getBytes().length;
            return size;
        }
    }

    /**
     * Producer interceptor with OpenTelemetry tracing
     */
    public class OpenTelemetryProducerInterceptor implements
ProducerInterceptor<String, Object> {

        @Override
        public ProducerRecord<String, Object> onSend(ProducerRecord<String,
Object> record) {

            // Create a span for the send operation
            Span span = tracer.spanBuilder("kafka.producer.send")
                .setAttribute("messaging.system", "kafka")
                .setAttribute("messaging.destination.name", record.topic())
                .setAttribute("messaging.operation", "send")
                .startSpan();

            try (Scope scope = span.makeCurrent()) {

                // Add span context to headers for trace propagation
                io.opentelemetry.context.Context currentContext =
io.opentelemetry.context.Context.current();

                // Inject trace context into Kafka headers
                TextMapSetter<Headers> setter = (headers, key, value) ->
                    headers.add(key, value.getBytes());
```

```
                openTelemetry.getPropagators().getTextMapPropagator()
                    .inject(currentContext, record.headers(), setter);

                // Add additional span attributes
                span.setAttribute("messaging.kafka.destination.partition",
                    record.partition() != null ? record.partition() : -1);
                span.setAttribute("messaging.message.payload_size_bytes",
                    estimateRecordSize(record));

                return record;

            } finally {
                span.end();
            }
        }

        @Override
        public void onAcknowledgement(RecordMetadata metadata, Exception
exception) {

            // Create span for acknowledgment
            Span span = tracer.spanBuilder("kafka.producer.acknowledgment")
                .setAttribute("messaging.system", "kafka")
                .startSpan();

            try (Scope scope = span.makeCurrent()) {

                if (exception == null && metadata != null) {
                    span.setAttribute("messaging.kafka.destination.partition",
metadata.partition());
                    span.setAttribute("messaging.kafka.message.offset",
metadata.offset());
                    span.setStatus(io.opentelemetry.api.trace.StatusCode.OK);
                } else {
                    span.setAttribute("error.message", exception != null ?
exception.getMessage() : "unknown");
                    span.setStatus(io.opentelemetry.api.trace.StatusCode.ERROR,
                        exception != null ? exception.getMessage() : "Send
failed");
                }

            } finally {
                span.end();
            }
        }

        @Override
        public void close() {
            log.info("Closing OpenTelemetry producer interceptor");
        }

        @Override
        public void configure(Map<String, ?> configs) {
            log.info("Configuring OpenTelemetry producer interceptor");
```

```
        }

        private long estimateRecordSize(ProducerRecord<String, Object> record) {
            long size = 0;
            if (record.key() != null) size += record.key().getBytes().length;
            if (record.value() != null) size +=
record.value().toString().getBytes().length;
            return size;
        }
    }

    /**
     * Consumer interceptor with OpenTelemetry tracing
     */
    public class OpenTelemetryConsumerInterceptor implements
ConsumerInterceptor<String, Object> {

        @Override
        public ConsumerRecords<String, Object> onConsume(ConsumerRecords<String,
Object> records) {

            for (ConsumerRecord<String, Object> record : records) {

                // Extract trace context from headers
                TextMapGetter<Headers> getter = new TextMapGetter<Headers>() {
                    @Override
                    public Iterable<String> keys(Headers headers) {
                        return () -> StreamSupport.stream(headers.spliterator(),
false)
                                .map(header -> header.key())
                                .iterator();
                    }

                    @Override
                    public String get(Headers headers, String key) {
                        Header header = headers.lastHeader(key);
                        return header != null ? new String(header.value()) : null;
                    }
                };

                io.opentelemetry.context.Context extractedContext =
                    openTelemetry.getPropagators().getTextMapPropagator()
                        .extract(io.opentelemetry.context.Context.current(),
record.headers(), getter);

                // Create consumer receive span
                Span span = tracer.spanBuilder("kafka.consumer.receive")
                    .setParent(extractedContext)
                    .setAttribute("messaging.system", "kafka")
                    .setAttribute("messaging.destination.name", record.topic())
                    .setAttribute("messaging.operation", "receive")
                    .setAttribute("messaging.kafka.source.partition",
record.partition())
                    .setAttribute("messaging.kafka.message.offset",
```

```
record.offset())
                    .setAttribute("messaging.message.payload_size_bytes",
estimateRecordSize(record))
                    .startSpan();

                // Store span in record headers for later use
                record.headers().add("otel-span-id",
span.getSpanContext().getSpanId().getBytes());
                record.headers().add("otel-trace-id",
span.getSpanContext().getTraceId().getBytes());

                span.end();
            }

            return records;
        }

        @Override
        public void onCommit(Map<TopicPartition, OffsetAndMetadata> offsets) {

            // Create span for commit operation
            Span span = tracer.spanBuilder("kafka.consumer.commit")
                .setAttribute("messaging.system", "kafka")
                .setAttribute("messaging.operation", "commit")
                .startSpan();

            try (Scope scope = span.makeCurrent()) {

                span.setAttribute("kafka.consumer.committed_offsets.count",
offsets.size());

                // Add details about committed offsets
                StringBuilder offsetDetails = new StringBuilder();
                offsets.forEach((tp, offsetMetadata) -> {
                    offsetDetails.append(String.format("%s:%d=%d ",
                        tp.topic(), tp.partition(), offsetMetadata.offset()));
                });

                span.setAttribute("kafka.consumer.committed_offsets.details",
                    offsetDetails.toString().trim());

            } finally {
                span.end();
            }
        }

        @Override
        public void close() {
            log.info("Closing OpenTelemetry consumer interceptor");
        }

        @Override
        public void configure(Map<String, ?> configs) {
            log.info("Configuring OpenTelemetry consumer interceptor");
```

```
        }

        private long estimateRecordSize(ConsumerRecord<String, Object> record) {
            long size = 0;
            if (record.key() != null) size += record.key().getBytes().length;
            if (record.value() != null) size +=
record.value().toString().getBytes().length;
            return size;
        }
    }
}

/**
 * Service with OpenTelemetry manual instrumentation
 */
@Service
@lombok.extern.slf4j.Slf4j
public class OpenTelemetryKafkaService {

    @Autowired
    private KafkaTemplate<String, Object> kafkaTemplate;

    @Autowired
    private OpenTelemetry openTelemetry;

    private final Tracer tracer;

    public OpenTelemetryKafkaService(OpenTelemetry openTelemetry) {
        this.tracer = openTelemetry.getTracer("kafka-service", "1.0.0");
    }

    /**
     * Send message with manual OpenTelemetry tracing
     */
    public void sendMessageWithOpenTelemetry(String topic, String key, Object
message) {

        Span span = tracer.spanBuilder("business.kafka.send")
            .setAttribute("messaging.system", "kafka")
            .setAttribute("messaging.destination.name", topic)
            .setAttribute("business.operation", "send.message")
            .setAttribute("message.type", message.getClass().getSimpleName())
            .startSpan();

        try (Scope scope = span.makeCurrent()) {

            log.info("Sending message with OpenTelemetry: topic={}, key={}",
topic, key);

            // Add business context to span
            span.setAttribute("business.message.processed", true);
            span.setAttribute("kafka.message.key", key);

            // Send message (will be automatically traced by interceptors)
```

```
            ListenableFuture<SendResult<String, Object>> future =
kafkaTemplate.send(topic, key, message);

            // Handle result
            future.addCallback(
                result -> {
                    span.setAttribute("kafka.send.success", true);
                    span.setAttribute("kafka.partition",
result.getRecordMetadata().partition());
                    span.setAttribute("kafka.offset",
result.getRecordMetadata().offset());
                    span.setStatus(io.opentelemetry.api.trace.StatusCode.OK);
                },
                failure -> {
                    span.setAttribute("kafka.send.success", false);
                    span.setAttribute("error.message", failure.getMessage());
                    span.setStatus(io.opentelemetry.api.trace.StatusCode.ERROR,
failure.getMessage());
                }
            );

        } catch (Exception e) {
            span.recordException(e);
            span.setStatus(io.opentelemetry.api.trace.StatusCode.ERROR,
e.getMessage());
            throw e;
        } finally {
            span.end();
        }
    }

    /**
     * Consume message with OpenTelemetry tracing
     */
    @KafkaListener(
        topics = "otel-events",
        groupId = "otel-consumer-group",
        containerFactory = "otelKafkaListenerContainerFactory"
    )
    public void consumeMessageWithOpenTelemetry(@Payload OtelEvent event,
                                                @Header(KafkaHeaders.RECEIVED_TOPIC)
String topic,

@Header(KafkaHeaders.RECEIVED_PARTITION) int partition,
                                                @Header(KafkaHeaders.OFFSET) long
offset,
                                                ConsumerRecord<String, Object>
consumerRecord) {

        // Extract trace context from record headers
        io.opentelemetry.context.Context extractedContext =
extractTraceContext(consumerRecord.headers());

        Span span = tracer.spanBuilder("business.kafka.consume")
```

```
            .setParent(extractedContext)
            .setAttribute("messaging.system", "kafka")
            .setAttribute("messaging.destination.name", topic)
            .setAttribute("messaging.kafka.source.partition", partition)
            .setAttribute("messaging.kafka.message.offset", offset)
            .setAttribute("business.operation", "consume.message")
            .setAttribute("event.type", event.getEventType())
            .setAttribute("event.id", event.getEventId())
            .startSpan();

        try (Scope scope = span.makeCurrent()) {

            log.info("Processing message with OpenTelemetry: eventId={}, topic={},
offset={}",
                event.getEventId(), topic, offset);

            // Process the event
            processOtelEvent(event);

            // Add processing results to span
            span.setAttribute("business.processing.successful", true);
            span.setAttribute("business.processing.duration_ms", 50);
            span.setStatus(io.opentelemetry.api.trace.StatusCode.OK);

        } catch (Exception e) {
            span.recordException(e);
            span.setAttribute("business.processing.successful", false);
            span.setStatus(io.opentelemetry.api.trace.StatusCode.ERROR,
e.getMessage());
            throw e;
        } finally {
            span.end();
        }
    }

    private io.opentelemetry.context.Context extractTraceContext(Headers headers)
{

        TextMapGetter<Headers> getter = new TextMapGetter<Headers>() {
            @Override
            public Iterable<String> keys(Headers headers) {
                return () -> StreamSupport.stream(headers.spliterator(), false)
                    .map(Header::key)
                    .iterator();
            }

            @Override
            public String get(Headers headers, String key) {
                Header header = headers.lastHeader(key);
                return header != null ? new String(header.value()) : null;
            }
        };

        return openTelemetry.getPropagators().getTextMapPropagator()
```

```
                .extract(io.opentelemetry.context.Context.current(), headers, getter);
        }

        private void processOtelEvent(OtelEvent event) {
            // Simulate event processing
            try {
                Thread.sleep(50);
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        }
    }

    // Supporting data classes
    @lombok.Data
    @lombok.AllArgsConstructor
    @lombok.NoArgsConstructor
    class OtelEvent {
        private String eventId;
        private String eventType;
        private String userId;
        private Instant timestamp;
    }
```

This completes Part 2 of the Spring Kafka Monitoring & Observability guide, covering health checks and distributed tracing. The guide continues with comparisons, best practices, and version highlights in the final part.