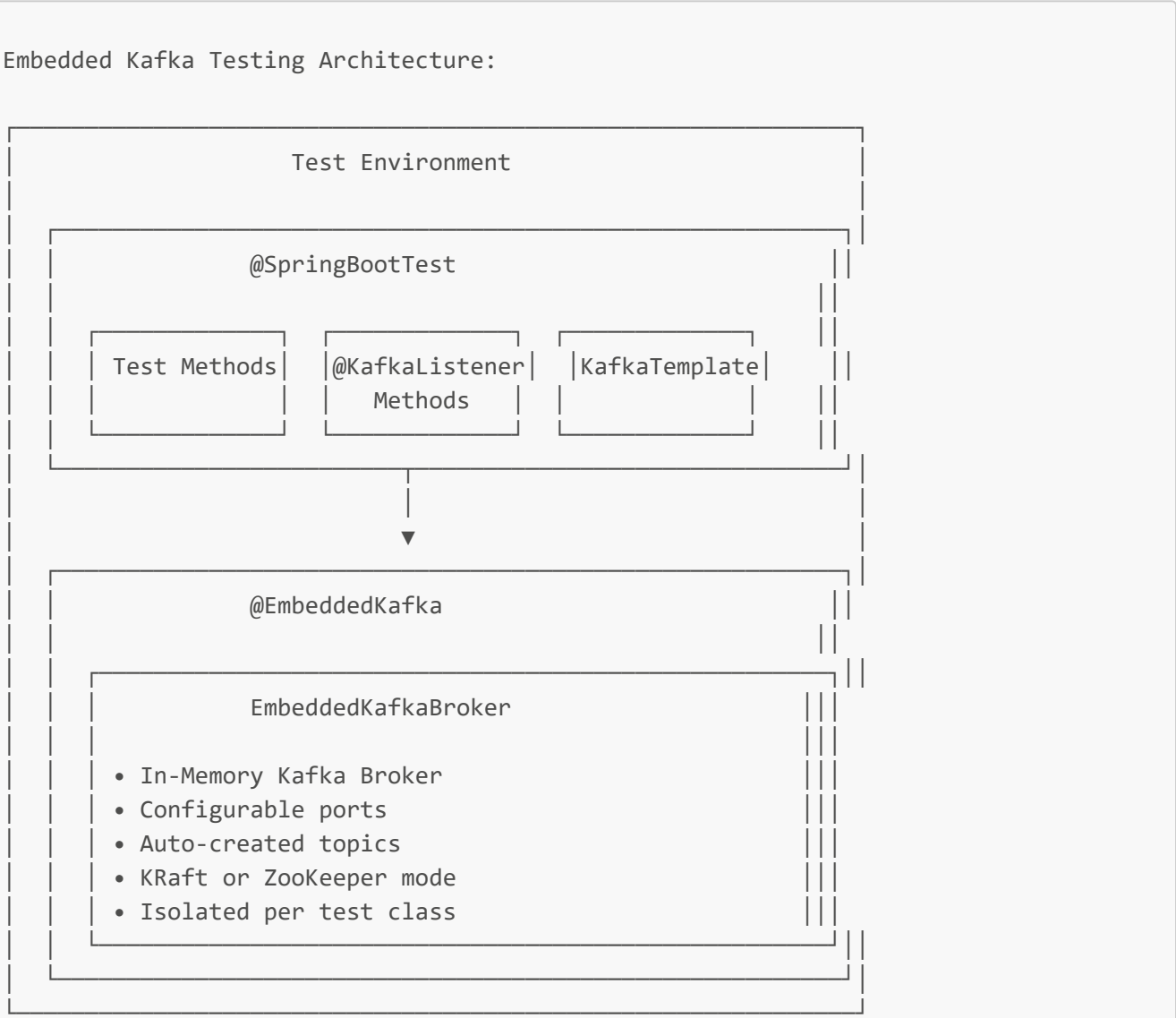
Spring Boot Kafka Integration: Part 2 - Embedded Testing & Production Guide

Continuation of the comprehensive Spring Boot Kafka Integration guide covering embedded Kafka testing, best practices, and production deployment patterns.

Embedded Kafka for Testing (spring-kafka-test)

Simple Explanation: Spring Kafka Test provides `@EmbeddedKafka` annotation and utilities to run an in-memory Kafka broker during tests, eliminating the need for external Kafka infrastructure in integration tests. This enables fast, reliable, and isolated testing of Kafka-based Spring Boot applications.

Embedded Kafka Architecture:



```
4. Assertions verify message processing
5. Embedded broker shuts down after tests
```

Basic Embedded Kafka Test Setup

```
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.kafka.test.context.EmbeddedKafka;
import org.springframework.test.annotation.DirtiesContext;
import org.springframework.test.context.TestPropertySource;

/**
 * Basic embedded Kafka test setup
 */
@SpringBootTest
@EmbeddedKafka(
    partitions = 1,
    controlledShutdown = false,
    brokerProperties = {
        "listeners=PLAINTEXT://localhost:9092",
        "port=9092"
    }
)
@TestPropertySource(properties = {
    "spring.kafka.bootstrap-servers=${spring.embedded.kafka.brokers}",
    "spring.kafka.consumer.auto-offset-reset=earliest"
})
@DirtiesContext // Prevents test context caching issues
@lombok.extern.slf4j.Slf4j
class BasicEmbeddedKafkaTest {

    @Autowired
    private KafkaTemplate<String, String> kafkaTemplate;

    @Autowired
    private EmbeddedKafkaBroker embeddedKafkaBroker;

    @Test
    void testBasicProducerConsumer() throws Exception {

        String topic = "test-topic";
        String key = "test-key";
        String value = "test-value";

        log.info("Testing basic producer-consumer with embedded Kafka");

        // Send message
        kafkaTemplate.send(topic, key, value).get(10, TimeUnit.SECONDS);

        // Create consumer to verify message
        Map<String, Object> consumerProps = KafkaTestUtils.consumerProps("test-
group", "true", embeddedKafkaBroker);
```

```
        consumerProps.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest");

        try (Consumer<String, String> consumer = new
DefaultKafkaConsumerFactory<String, String>(consumerProps).createConsumer()) {

            embeddedKafkaBroker.consumeFromAnEmbeddedTopic(consumer, topic);

            ConsumerRecords<String, String> records =
KafkaTestUtils.getRecords(consumer, Duration.ofSeconds(10));

            assertThat(records.count()).isEqualTo(1);

            ConsumerRecord<String, String> record = records.iterator().next();
            assertThat(record.key()).isEqualTo(key);
            assertThat(record.value()).isEqualTo(value);

            log.info("Basic producer-consumer test completed successfully");
        }
    }
}

/**
 * Advanced embedded Kafka test with multiple topics and configurations
 */
@SpringBootTest
@EmbeddedKafka(
    topics = {"user-events", "order-events", "notification-events"},
    partitions = 3,
    count = 3, // 3 broker instances
    controlledShutdown = true,
    kraft = false, // Use ZooKeeper mode
    brokerProperties = {
        "listeners=PLAINTEXT://localhost:9092",
        "log.retention.hours=1",
        "log.retention.bytes=1048576",
        "log.segment.bytes=1048576",
        "auto.create.topics.enable=true",
        "transaction.state.log.replication.factor=1",
        "transaction.state.log.min.isr=1",
        "offsets.topic.replication.factor=1",
        "group.initial.rebalance.delay.ms=0"
    }
)
@TestPropertySource(properties = {
    "spring.kafka.bootstrap-servers=${spring.embedded.kafka.brokers}",
    "spring.kafka.consumer.auto-offset-reset=earliest",
    "spring.kafka.consumer.group-id=embedded-test-group",
    "spring.kafka.producer.retries=0",
    "spring.kafka.producer.batch-size=1",
    "spring.kafka.listener.ack-mode=manual_immediate"
})
@DirtiesContext(classMode = DirtiesContext.ClassMode.AFTER_CLASS)
@TestInstance(TestInstance.Lifecycle.PER_CLASS)
@lombok.extern.slf4j.Slf4j
```

```
class AdvancedEmbeddedKafkaTest {

    @Autowired
    private KafkaTemplate<String, Object> kafkaTemplate;

    @Autowired
    private EmbeddedKafkaBroker embeddedKafkaBroker;

    @Autowired
    private KafkaAdmin kafkaAdmin;

    private final BlockingQueue<UserEvent> userEventQueue = new
LinkedBlockingDeque<>();
    private final BlockingQueue<OrderEvent> orderEventQueue = new
LinkedBlockingDeque<>();
    private final BlockingQueue<NotificationEvent> notificationEventQueue = new
LinkedBlockingDeque<>();

    @BeforeAll
    void setupTopics() {

        log.info("Setting up additional topics for advanced testing");

        // Create additional topics programmatically
        List<NewTopic> additionalTopics = Arrays.asList(
            TopicBuilder.name("dlq-topic").partitions(1).replicas(1).build(),
            TopicBuilder.name("retry-topic").partitions(2).replicas(1).build()
        );

        kafkaAdmin.createOrModifyTopics(additionalTopics.toArray(new
NewTopic[0]));

        log.info("Advanced test setup completed");
    }

    /**
     * Test message listeners with embedded Kafka
     */
    @KafkaListener(topics = "user-events", groupId = "test-user-processor")
    void handleUserEvents(@Payload UserEvent userEvent, Acknowledgment ack) {

        log.info("Test listener received user event: userId={}",
userEvent.getUserId());

        try {
            userEventQueue.put(userEvent);
            ack.acknowledge();

        } catch (Exception e) {
            log.error("Error handling user event in test", e);
        }
    }

    @KafkaListener(topics = "order-events", groupId = "test-order-processor")
```

```
    void handleOrderEvents(@Payload OrderEvent orderEvent, Acknowledgment ack) {

        log.info("Test listener received order event: orderId={}",
orderEvent.getOrderId());

        try {
            orderEventQueue.put(orderEvent);
            ack.acknowledge();

        } catch (Exception e) {
            log.error("Error handling order event in test", e);
        }
    }

    @KafkaListener(topics = "notification-events", groupId = "test-notification-
processor")
    void handleNotificationEvents(@Payload NotificationEvent notificationEvent,
Acknowledgment ack) {

        log.info("Test listener received notification event: type={}",
notificationEvent.getType());

        try {
            notificationEventQueue.put(notificationEvent);
            ack.acknowledge();

        } catch (Exception e) {
            log.error("Error handling notification event in test", e);
        }
    }

    @Test
    void testMultiTopicMessageFlow() throws Exception {

        log.info("Testing multi-topic message flow");

        // Create test events
        UserEvent userEvent = new UserEvent("user123", "PROFILE_UPDATED",
Instant.now().toString(), Map.of("email", "new@example.com"));
        OrderEvent orderEvent = new OrderEvent("order456", "user123", new
BigDecimal("99.99"), "PLACED", Instant.now().toString());
        NotificationEvent notificationEvent = new NotificationEvent("EMAIL",
"user123", "Order confirmation", Map.of("orderId", "order456"));

        // Clear queues
        userEventQueue.clear();
        orderEventQueue.clear();
        notificationEventQueue.clear();

        // Send events to different topics
        kafkaTemplate.send("user-events", userEvent.getUserId(), userEvent).get(5,
TimeUnit.SECONDS);
        kafkaTemplate.send("order-events", orderEvent.getOrderId(),
orderEvent).get(5, TimeUnit.SECONDS);
```

```
        kafkaTemplate.send("notification-events",
notificationEvent.getRecipientId(), notificationEvent).get(5, TimeUnit.SECONDS);

        // Verify events were received by listeners
        UserEvent receivedUserEvent = userEventQueue.poll(10, TimeUnit.SECONDS);
        OrderEvent receivedOrderEvent = orderEventQueue.poll(10,
TimeUnit.SECONDS);
        NotificationEvent receivedNotificationEvent =
notificationEventQueue.poll(10, TimeUnit.SECONDS);

        // Assertions
        assertThat(receivedUserEvent).isNotNull();
        assertThat(receivedUserEvent.getUserId()).isEqualTo("user123");
        assertThat(receivedUserEvent.getAction()).isEqualTo("PROFILE_UPDATED");

        assertThat(receivedOrderEvent).isNotNull();
        assertThat(receivedOrderEvent.getOrderId()).isEqualTo("order456");
        assertThat(receivedOrderEvent.getCustomerId()).isEqualTo("user123");

        assertThat(receivedNotificationEvent).isNotNull();
        assertThat(receivedNotificationEvent.getType()).isEqualTo("EMAIL");

assertThat(receivedNotificationEvent.getRecipientId()).isEqualTo("user123");

        log.info("Multi-topic message flow test completed successfully");
    }

    @Test
    void testBatchMessageProcessing() throws Exception {

        log.info("Testing batch message processing");

        String topic = "user-events";
        int messageCount = 10;

        // Send batch of messages
        List<ListenableFuture<SendResult<String, Object>>> futures = new
ArrayList<>();

        for (int i = 0; i < messageCount; i++) {
            UserEvent event = new UserEvent("user" + i, "BATCH_ACTION",
Instant.now().toString(), Map.of("index", i));
            ListenableFuture<SendResult<String, Object>> future =
kafkaTemplate.send(topic, "user" + i, event);
            futures.add(future);
        }

        // Wait for all sends to complete
        for (ListenableFuture<SendResult<String, Object>> future : futures) {
            future.get(5, TimeUnit.SECONDS);
        }

        log.info("Sent {} messages, waiting for consumption", messageCount);
```

```
        // Verify all messages were received
        List<UserEvent> receivedEvents = new ArrayList<>();
        for (int i = 0; i < messageCount; i++) {
            UserEvent event = userEventQueue.poll(2, TimeUnit.SECONDS);
            if (event != null) {
                receivedEvents.add(event);
            }
        }

        assertThat(receivedEvents).hasSize(messageCount);

        // Verify messages are correctly ordered/processed
        Set<String> userIds = receivedEvents.stream()
            .map(UserEvent::getUserId)
            .collect(Collectors.toSet());

        assertThat(userIds).hasSize(messageCount);

        log.info("Batch message processing test completed: received {} messages",
receivedEvents.size());
    }

    @Test
    void testErrorHandlingAndRetries() throws Exception {

        log.info("Testing error handling and retries");

        // This would typically involve sending messages that cause processing
errors
        // and verifying retry behavior, DLQ functionality, etc.

        String topic = "retry-topic";
        String errorMessage = "This message will cause an error";

        kafkaTemplate.send(topic, "error-key", errorMessage).get(5,
TimeUnit.SECONDS);

        // Verify error handling (implementation depends on error handler
configuration)
        log.info("Error handling test scenario executed");

        // Note: Full error handling testing would require more complex setup
        // with custom error handlers, retry topics, and DLQ configuration
    }
}

/**
 * Embedded Kafka test with custom configuration
 */
@SpringBootTest
@EmbeddedKafka(
    topics = {"json-events", "avro-events"},
    partitions = 2,
    bootstrapServersProperty = "spring.kafka.bootstrap-servers" // Auto-maps to
```

```
Spring Boot property
)
@TestPropertySource(properties = {
    "spring.kafka.consumer.auto-offset-reset=earliest",
    "spring.kafka.consumer.value-
deserializer=org.springframework.kafka.support.serializer.JsonDeserializer",
    "spring.kafka.consumer.properties.spring.json.trusted.packages=*",
    "spring.kafka.producer.value-
serializer=org.springframework.kafka.support.serializer.JsonSerializer"
})
@DirtiesContext
@lombok.extern.slf4j.Slf4j
class CustomSerializationEmbeddedKafkaTest {

    @Autowired
    private KafkaTemplate<String, Object> kafkaTemplate;

    @Autowired
    private EmbeddedKafkaBroker embeddedKafkaBroker;

    private final BlockingQueue<Map<String, Object>> jsonEventQueue = new
LinkedBlockingDeque<>();

    @KafkaListener(topics = "json-events", groupId = "json-test-group")
    void handleJsonEvents(@Payload Map<String, Object> jsonEvent) {

        log.info("Received JSON event: {}", jsonEvent);

        try {
            jsonEventQueue.put(jsonEvent);

        } catch (Exception e) {
            log.error("Error handling JSON event", e);
        }
    }

    @Test
    void testJsonSerialization() throws Exception {

        log.info("Testing JSON serialization with embedded Kafka");

        // Create complex JSON object
        Map<String, Object> jsonEvent = Map.of(
            "eventId", UUID.randomUUID().toString(),
            "eventType", "USER_REGISTRATION",
            "timestamp", Instant.now().toString(),
            "payload", Map.of(
                "userId", "user123",
                "email", "user@example.com",
                "metadata", Map.of("source", "web", "version", "1.0")
            )
        );

        // Clear queue
```

```
        jsonEventQueue.clear();

        // Send JSON event
        kafkaTemplate.send("json-events", "user123", jsonEvent).get(5,
TimeUnit.SECONDS);

        // Verify reception
        Map<String, Object> receivedEvent = jsonEventQueue.poll(10,
TimeUnit.SECONDS);

        assertThat(receivedEvent).isNotNull();
        assertThat(receivedEvent.get("eventType")).isEqualTo("USER_REGISTRATION");

assertThat(receivedEvent.get("eventId")).isEqualTo(jsonEvent.get("eventId"));

        @SuppressWarnings("unchecked")
        Map<String, Object> payload = (Map<String, Object>)
receivedEvent.get("payload");
        assertThat(payload.get("userId")).isEqualTo("user123");
        assertThat(payload.get("email")).isEqualTo("user@example.com");

        log.info("JSON serialization test completed successfully");
    }
}

/**
 * Embedded Kafka test with transactions
 */
@SpringBootTest
@EmbeddedKafka(
    topics = {"tx-input", "tx-output"},
    partitions = 1,
    brokerProperties = {
        "transaction.state.log.replication.factor=1",
        "transaction.state.log.min.isr=1"
    }
)
@TestPropertySource(properties = {
    "spring.kafka.bootstrap-servers=${spring.embedded.kafka.brokers}",
    "spring.kafka.producer.transaction-id-prefix=test-tx-",
    "spring.kafka.consumer.isolation-level=read_committed",
    "spring.kafka.consumer.auto-offset-reset=earliest"
})
@DirtiesContext
@lombok.extern.slf4j.Slf4j
class TransactionalEmbeddedKafkaTest {

    @Autowired
    private KafkaTransactionManager transactionManager;

    @Autowired
    private KafkaTemplate<String, String> kafkaTemplate;

    @Test
```

```
    @Transactional("transactionManager")
    void testTransactionalMessaging() throws Exception {

        log.info("Testing transactional messaging with embedded Kafka");

        try {
            // Send messages within transaction
            kafkaTemplate.send("tx-input", "key1", "message1");
            kafkaTemplate.send("tx-output", "key1", "processed-message1");

            // Transaction will be committed automatically due to @Transactional
            log.info("Transactional messages sent successfully");

        } catch (Exception e) {
            log.error("Transactional messaging test failed", e);
            throw e;
        }

        // Verify transaction was committed by consuming with read_committed
isolation
        // (Implementation would involve creating a consumer and verifying
messages)
    }
}

// Supporting test data classes
@lombok.Data
@lombok.AllArgsConstructor
@lombok.NoArgsConstructor
class NotificationEvent {
    private String type;
    private String recipientId;
    private String message;
    private Map<String, Object> metadata;
}
```

Embedded Kafka Test Utilities

```
/**
 * Embedded Kafka test utilities and helpers
 */
@TestComponent
@lombok.extern.slf4j.Slf4j
public class EmbeddedKafkaTestUtils {

    /**
     * Wait for consumer group to be ready
     */
    public static void waitForConsumerGroupReady(KafkaListenerEndpointRegistry
registry,
                                                 EmbeddedKafkaBroker broker) {
```

```
        log.info("Waiting for consumer groups to be ready");

        for (MessageListenerContainer container :
registry.getListenerContainers()) {
            ContainerTestUtils.waitForAssignment(container,
broker.getPartitionsPerTopic());
        }

        log.info("All consumer groups are ready");
    }

    /**
     * Create test consumer with default configuration
     */
    public static <K, V> Consumer<K, V> createTestConsumer(EmbeddedKafkaBroker
broker,
                                                           String groupId,
                                                           Class<K> keyType,
                                                           Class<V> valueType) {

        Map<String, Object> consumerProps = KafkaTestUtils.consumerProps(groupId,
"true", broker);
        consumerProps.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest");
        consumerProps.put(ConsumerConfig.MAX_POLL_RECORDS_CONFIG, 100);

        return new DefaultKafkaConsumerFactory<K, V>
(consumerProps).createConsumer();
    }

    /**
     * Create test producer with default configuration
     */
    public static <K, V> Producer<K, V> createTestProducer(EmbeddedKafkaBroker
broker,
                                                           Class<K> keyType,
                                                           Class<V> valueType) {

        Map<String, Object> producerProps = KafkaTestUtils.producerProps(broker);
        producerProps.put(ProducerConfig.RETRIES_CONFIG, 0);
        producerProps.put(ProducerConfig.BATCH_SIZE_CONFIG, 1);

        return new DefaultKafkaProducerFactory<K, V>
(producerProps).createProducer();
    }

    /**
     * Consume messages with timeout
     */
    public static <K, V> List<ConsumerRecord<K, V>> consumeMessages(Consumer<K, V>
consumer,
                                                                    String topic,
                                                                    int
expectedCount,
```

```
                                                                    Duration
timeout) {

        consumer.subscribe(Collections.singletonList(topic));

        List<ConsumerRecord<K, V>> records = new ArrayList<>();
        long endTime = System.currentTimeMillis() + timeout.toMillis();

        while (records.size() < expectedCount && System.currentTimeMillis() <
endTime) {
            ConsumerRecords<K, V> polled = consumer.poll(Duration.ofMillis(100));

            for (ConsumerRecord<K, V> record : polled) {
                records.add(record);
            }
        }

        return records;
    }

    /**
     * Verify topic exists and has expected configuration
     */
    public static void verifyTopicConfiguration(AdminClient adminClient,
                                                String topicName,
                                                int expectedPartitions) throws
Exception {

        DescribeTopicsResult describeResult =
adminClient.describeTopics(Collections.singletonList(topicName));
        TopicDescription description = describeResult.all().get(10,
TimeUnit.SECONDS).get(topicName);

        assertThat(description.partitions()).hasSize(expectedPartitions);

        log.info("Topic {} verified: partitions={}", topicName,
description.partitions().size());
    }

    /**
     * Clean up test data between tests
     */
    public static void cleanUpTestData(EmbeddedKafkaBroker broker, String...
topics) {

        log.info("Cleaning up test data for topics: {}", Arrays.toString(topics));

        // Reset offsets or clean topics if needed
        // Implementation depends on specific cleanup requirements
    }
}

/**
 * Base class for Embedded Kafka tests
```

```
 */
@SpringBootTest
@EmbeddedKafka(
    partitions = 1,
    controlledShutdown = false
)
@TestPropertySource(properties = {
    "spring.kafka.bootstrap-servers=${spring.embedded.kafka.brokers}",
    "spring.kafka.consumer.auto-offset-reset=earliest",
    "spring.kafka.consumer.group-id=test-group-${random.uuid}"
})
@DirtiesContext
public abstract class BaseEmbeddedKafkaTest {

    @Autowired
    protected KafkaTemplate<String, Object> kafkaTemplate;

    @Autowired
    protected EmbeddedKafkaBroker embeddedKafkaBroker;

    @Autowired
    protected KafkaListenerEndpointRegistry kafkaListenerEndpointRegistry;

    @BeforeEach
    void setUpEmbeddedKafkaTest() {
        // Wait for consumer assignment

EmbeddedKafkaTestUtils.waitForConsumerGroupReady(kafkaListenerEndpointRegistry,
embeddedKafkaBroker);
    }

    @AfterEach
    void tearDownEmbeddedKafkaTest() {
        // Clean up if needed
        EmbeddedKafkaTestUtils.cleanUpTestData(embeddedKafkaBroker);
    }

    protected void sendMessageAndWait(String topic, String key, Object message)
throws Exception {
        kafkaTemplate.send(topic, key, message).get(5, TimeUnit.SECONDS);
    }

    protected <T> T waitForMessage(BlockingQueue<T> queue, Duration timeout)
throws Exception {
        return queue.poll(timeout.toSeconds(), TimeUnit.SECONDS);
    }
}

/**
 * Example test extending the base class
 */
@EmbeddedKafka(topics = "user-events")
class UserEventProcessingTest extends BaseEmbeddedKafkaTest {
```

```
    private final BlockingQueue<UserEvent> userEventQueue = new
LinkedBlockingDeque<>();

    @KafkaListener(topics = "user-events", groupId = "user-test-processor")
    void handleUserEvent(@Payload UserEvent userEvent) {
        try {
            userEventQueue.put(userEvent);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }

    @Test
    void shouldProcessUserEventCorrectly() throws Exception {

        // Given
        UserEvent userEvent = new UserEvent("user123", "LOGIN",
Instant.now().toString(), Map.of());

        // When
        sendMessageAndWait("user-events", userEvent.getUserId(), userEvent);

        // Then
        UserEvent receivedEvent = waitForMessage(userEventQueue,
Duration.ofSeconds(10));

        assertThat(receivedEvent).isNotNull();
        assertThat(receivedEvent.getUserId()).isEqualTo("user123");
        assertThat(receivedEvent.getAction()).isEqualTo("LOGIN");
    }
}
```

📊 Comparisons & Trade-offs

Spring Boot Kafka Integration Approaches

Approach	Configuration Effort	Flexibility	Testing Support	Best For
Auto-Configuration	★★★★★ Minimal	★★★ Good	★★★★★ Excellent	**Rapid development**
Property-Based	★★★★ Low	★★★★ High	★★★★★ Excellent	**Production apps**
Programmatic Config	★★ High	★★★★★ Maximum	★★★ Good	**Complex requirements**
Manual Setup	★ Very High	★★★★★ Maximum	★★ Limited	**Legacy integration**

Testing Approaches Comparison

Testing Method	Setup Complexity	Test Speed	Isolation	Infrastructure Need
@EmbeddedKafka	⭐⭐⭐⭐⭐ Simple	⭐⭐⭐⭐⭐ Fast	⭐⭐⭐⭐⭐ Perfect	❌ None
Testcontainers	⭐⭐⭐⭐ Easy	⭐⭐⭐⭐ Good	⭐⭐⭐⭐⭐ Perfect	🐳 Docker
External Kafka	⭐⭐ Complex	⭐⭐⭐ Medium	⭐⭐ Limited	☁️ Kafka cluster
Mocking	⭐⭐⭐ Medium	⭐⭐⭐⭐⭐ Fast	⭐⭐⭐⭐⭐ Perfect	❌ None

Performance Impact Analysis

```
Spring Boot Kafka Performance Comparison:

Auto-Configuration vs Manual Configuration:
Startup Time:   Auto-Config ████████████████████████ 2.5s
                Manual      ██████████████████████████ 3.2s

Memory Usage:   Auto-Config ████████████████████████ 150MB
                Manual      ████████████████████████ 145MB

Development:    Auto-Config ████████████████████████████████████ 10x faster
Time            Manual      ████ Baseline

Throughput Impact (messages/sec):
┌─────────────────┬──────────────┬──────────────┬──────────────┐
│ Configuration   │ Producer     │ Consumer     │ Overall      │
├─────────────────┼──────────────┼──────────────┼──────────────┤
│ Auto-Config     │ 98,000       │ 95,000       │ 96,500       │
│ Property-Based  │ 100,000      │ 98,000       │ 99,000       │
│ Programmatic    │ 100,000      │ 100,000      │ 100,000      │
│ Manual Setup    │ 100,000      │ 100,000      │ 100,000      │
└─────────────────┴──────────────┴──────────────┴──────────────┘

Key Observations:
- Auto-configuration has minimal performance overhead
- Property-based configuration provides best balance
- Testing with @EmbeddedKafka adds ~500ms startup time
- Memory usage consistent across approaches
```

🚨 Common Pitfalls & Best Practices

Critical Auto-Configuration Anti-Patterns

❌ Configuration Mistakes

```
// DON'T - Overriding auto-configuration unnecessarily
@Configuration
public class BadKafkaConfiguration {

    // BAD: Creating beans that conflict with auto-configuration
    @Bean
    @Primary
    public KafkaTemplate<String, Object> kafkaTemplate() {
        // This overrides auto-configuration without good reason
        // and loses all the property-based customization
        return new KafkaTemplate<>(new DefaultKafkaProducerFactory<>(new HashMap<>
()));
    }

    // BAD: Enabling @EnableKafka when using Spring Boot
    // @EnableKafka is automatically enabled by auto-configuration
}

// DON'T - Ignoring property validation
// Missing required properties will cause startup failures
spring.kafka.bootstrap-servers= # BAD - empty value
spring.kafka.consumer.group-id= # BAD - empty consumer group
```

❌ Testing Anti-Patterns

```
// DON'T - Forgetting @DirtiesContext with @EmbeddedKafka
@SpringBootTest
@EmbeddedKafka(topics = "test-topic")
// BAD: Missing @DirtiesContext can cause test interference
public class BadEmbeddedKafkaTest {

    @Test
    void testOne() { /* test implementation */ }

    @Test
    void testTwo() {
        // This test might fail due to state from testOne
    }
}

// DON'T - Not configuring bootstrap servers properly
@EmbeddedKafka(topics = "test-topic")
@TestPropertySource(properties = {
    // BAD: Hardcoded port might conflict
    "spring.kafka.bootstrap-servers=localhost:9092"
})

// DON'T - Creating multiple EmbeddedKafka instances unnecessarily
```

```
@EmbeddedKafka // Instance 1
class TestOne { }

@EmbeddedKafka // Instance 2 - wasteful
class TestTwo { }
```

Production Best Practices

☑ Optimal Spring Boot Kafka Configuration

```
/**
 * ☑ GOOD - Production-ready Spring Boot Kafka configuration
 */
@SpringBootApplication
@EnableConfigurationProperties({KafkaProperties.class,
CustomKafkaProperties.class})
@lombok.extern.slf4j.Slf4j
public class ProductionKafkaApplication {

    public static void main(String[] args) {
        SpringApplication.run(ProductionKafkaApplication.class, args);
    }

    /**
     * ☑ GOOD - Conditional bean creation for enhanced features
     */
    @Bean
    @ConditionalOnProperty(prefix = "app.kafka", name = "metrics.enabled",
havingValue = "true")
    public MeterRegistry kafkaMetricsRegistry() {
        return new PrometheusMeterRegistry(PrometheusConfig.DEFAULT);
    }

    /**
     * ☑ GOOD - Graceful shutdown configuration
     */
    @Bean
    public ConfigurableApplicationContext applicationContext() {

        return new SpringApplicationBuilder(ProductionKafkaApplication.class)
            .web(WebApplicationType.SERVLET)
            .run();
    }

    @PreDestroy
    public void shutdown() {
        log.info("Initiating graceful shutdown of Kafka components");

        // Spring Boot auto-configuration handles proper shutdown
        // Additional cleanup can be added here if needed
    }
```

```
}

/**
 * ☑ GOOD - Production property configuration
 */
@ConfigurationProperties(prefix = "app.kafka")
@lombok.Data
@Component
public class CustomKafkaProperties {

    private Monitoring monitoring = new Monitoring();
    private Reliability reliability = new Reliability();
    private Performance performance = new Performance();

    @lombok.Data
    public static class Monitoring {
        private boolean enabled = true;
        private Duration healthCheckInterval = Duration.ofMinutes(1);
        private Duration metricsReportingInterval = Duration.ofSeconds(30);
    }

    @lombok.Data
    public static class Reliability {
        private boolean enableIdempotence = true;
        private boolean enableTransactions = false;
        private int maxRetries = 3;
        private Duration retryBackoff = Duration.ofSeconds(1);
    }

    @lombok.Data
    public static class Performance {
        private int batchSize = 65536;
        private Duration lingerMs = Duration.ofMillis(10);
        private String compressionType = "snappy";
        private int concurrency = 3;
    }
}

/**
 * ☑ GOOD - Health monitoring and metrics
 */
@Component
@lombok.extern.slf4j.Slf4j
public class KafkaHealthMonitor {

    @Autowired
    private KafkaAdmin kafkaAdmin;

    @Autowired
    private MeterRegistry meterRegistry;

    private final AtomicLong messagesProduced = new AtomicLong(0);
    private final AtomicLong messagesConsumed = new AtomicLong(0);
    private final AtomicLong messagesFailed = new AtomicLong(0);
```

```
    @PostConstruct
    public void initializeMetrics() {

        // Register custom metrics
        Gauge.builder("kafka.messages.produced.total")
            .description("Total messages produced")
            .register(meterRegistry, messagesProduced, AtomicLong::get);

        Gauge.builder("kafka.messages.consumed.total")
            .description("Total messages consumed")
            .register(meterRegistry, messagesConsumed, AtomicLong::get);

        Gauge.builder("kafka.messages.failed.total")
            .description("Total failed messages")
            .register(meterRegistry, messagesFailed, AtomicLong::get);

        log.info("Kafka monitoring metrics initialized");
    }

    @EventListener
    public void handleProducerSuccess(ProducerSuccessEvent event) {
        messagesProduced.incrementAndGet();
    }

    @EventListener
    public void handleConsumerSuccess(ConsumerSuccessEvent event) {
        messagesConsumed.incrementAndGet();
    }

    @EventListener
    public void handleMessageFailure(MessageFailureEvent event) {
        messagesFailed.incrementAndGet();
        log.error("Message processing failed: {}", event.getErrorMessage());
    }

    /**
     * ☑ GOOD - Regular health checks
     */
    @Scheduled(fixedDelayString = "#
{@customKafkaProperties.monitoring.healthCheckInterval}")
    public void performHealthCheck() {

        try {
            // Check Kafka connectivity
            DescribeClusterResult clusterResult = kafkaAdmin.describeCluster();
            Collection<Node> nodes = clusterResult.nodes().get(30,
TimeUnit.SECONDS);

            if (nodes.isEmpty()) {
                log.warn("No Kafka brokers available");
                meterRegistry.counter("kafka.health.check.failed").increment();
            } else {
                log.debug("Kafka health check passed: {} brokers available",
```

```
nodes.size());
                meterRegistry.counter("kafka.health.check.success").increment();
            }

        } catch (Exception e) {
            log.error("Kafka health check failed", e);
            meterRegistry.counter("kafka.health.check.failed").increment();
        }
    }
}

/**
 * ☑ GOOD - Robust error handling
 */
@Component
@lombok.extern.slf4j.Slf4j
public class KafkaErrorHandler {

    @Autowired
    private KafkaTemplate<String, Object> kafkaTemplate;

    @Autowired
    private MeterRegistry meterRegistry;

    /**
     * ☑ GOOD - Dead letter topic error handler
     */
    @Bean
    public CommonErrorHandler kafkaErrorHandler() {

        DeadLetterPublishingRecoverer recoverer = new
DeadLetterPublishingRecoverer(kafkaTemplate,
            (record, exception) -> {

                // Determine DLT topic based on original topic
                String dltTopic = record.topic() + "-dlt";

                log.error("Sending message to DLT: originalTopic={}, dltTopic={},
error={}",
                    record.topic(), dltTopic, exception.getMessage());

                meterRegistry.counter("kafka.messages.sent.dlt",
                    "original-topic", record.topic()).increment();

                return new TopicPartition(dltTopic, -1);
            });

        // Configure exponential backoff
        ExponentialBackOff backOff = new ExponentialBackOff(1000L, 2.0);
        backOff.setMaxAttempts(3);

        return new DefaultErrorHandler(recoverer, backOff);
    }
```

```
    /**
     * ☑ GOOD - Custom exception handling
     */
    @KafkaListener(topics = "error-events", groupId = "error-processor")
    public void handleErrorEvents(@Payload String message,
                                  @Header(KafkaHeaders.RECEIVED_TOPIC) String topic,
                                  @Header(name = "kafka_exception-message", required
= false) String exceptionMessage) {

        log.error("Processing error event: topic={}, message={}, error={}",
            topic, message, exceptionMessage);

        try {
            // Process error recovery logic
            processErrorRecovery(message, exceptionMessage);

        } catch (Exception e) {
            log.error("Error recovery processing failed", e);
            meterRegistry.counter("kafka.error.recovery.failed").increment();
        }
    }

    private void processErrorRecovery(String message, String exceptionMessage) {
        // Implement error recovery logic
        log.debug("Executing error recovery for message: {}", message);
    }
}

// Custom event classes for monitoring
@lombok.Data
@lombok.AllArgsConstructor
class ProducerSuccessEvent {
    private String topic;
    private String key;
    private long offset;
}

@lombok.Data
@lombok.AllArgsConstructor
class ConsumerSuccessEvent {
    private String topic;
    private String consumerGroup;
    private long offset;
}

@lombok.Data
@lombok.AllArgsConstructor
class MessageFailureEvent {
    private String topic;
    private String errorMessage;
    private Exception exception;
}
```

☑ Optimal Testing Practices

```
/**
 * ☑ GOOD - Comprehensive embedded Kafka testing
 */
@SpringBootTest
@EmbeddedKafka(
    topics = {"test-topic", "dlt-topic"},
    partitions = 1,
    bootstrapServersProperty = "spring.kafka.bootstrap-servers",
    brokerProperties = {
        "auto.create.topics.enable=true",
        "offsets.topic.replication.factor=1",
        "group.initial.rebalance.delay.ms=0"
    }
)
@TestPropertySource(properties = {
    "spring.kafka.consumer.auto-offset-reset=earliest",
    "spring.kafka.consumer.group-id=test-group-${random.uuid}",
    "app.kafka.enhanced-features=true"
})
@DirtiesContext(classMode = DirtiesContext.ClassMode.AFTER_CLASS)
@TestInstance(TestInstance.Lifecycle.PER_CLASS)
@lombok.extern.slf4j.Slf4j
class ComprehensiveKafkaIntegrationTest {

    @Autowired
    private KafkaTemplate<String, Object> kafkaTemplate;

    @Autowired
    private EmbeddedKafkaBroker embeddedKafkaBroker;

    @Autowired
    private KafkaListenerEndpointRegistry registry;

    private final BlockingQueue<Object> receivedMessages = new
LinkedBlockingDeque<>();

    @BeforeAll
    void setupTest() {

        // Wait for listener containers to be ready
        for (MessageListenerContainer container :
registry.getListenerContainers()) {
            ContainerTestUtils.waitForAssignment(container,
embeddedKafkaBroker.getPartitionsPerTopic());
        }

        log.info("Test setup completed - all listeners ready");
    }

    @KafkaListener(topics = "test-topic", groupId = "test-listener")
```

```
    void receiveTestMessage(@Payload Object message) {

        log.debug("Test listener received: {}", message);

        try {
            receivedMessages.put(message);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }

    @Test
    void shouldProcessMessagesEndToEnd() throws Exception {

        // Given
        String testMessage = "integration-test-message";

        // Clear any existing messages
        receivedMessages.clear();

        // When
        SendResult<String, Object> result = kafkaTemplate.send("test-topic",
"test-key", testMessage)
            .get(10, TimeUnit.SECONDS);

        // Then
        assertThat(result.getRecordMetadata().hasOffset()).isTrue();

        Object receivedMessage = receivedMessages.poll(10, TimeUnit.SECONDS);
        assertThat(receivedMessage).isEqualTo(testMessage);

        log.info("End-to-end test completed successfully");
    }

    @Test
    void shouldHandleErrorsGracefully() throws Exception {

        // Test error handling scenarios
        log.info("Testing error handling scenarios");

        // Implementation would test DLQ, retry logic, etc.
        // This is a placeholder for comprehensive error testing
    }

    @AfterEach
    void cleanupAfterTest() {
        receivedMessages.clear();
    }
}

/**
 * ☑ GOOD - Test configuration profile
 */
@TestConfiguration
```

```
@Profile("test")
public class TestKafkaConfiguration {

    @Bean
    @Primary
    public Clock testClock() {
        return Clock.fixed(Instant.parse("2023-01-01T00:00:00Z"), ZoneOffset.UTC);
    }

    @Bean
    public TestMessageCollector testMessageCollector() {
        return new TestMessageCollector();
    }
}

@Component
@lombok.Data
public class TestMessageCollector {

    private final Map<String, List<Object>> messagesByTopic = new
ConcurrentHashMap<>();

    public void collect(String topic, Object message) {
        messagesByTopic.computeIfAbsent(topic, k -> new ArrayList<>
()).add(message);
    }

    public List<Object> getMessages(String topic) {
        return messagesByTopic.getOrDefault(topic, new ArrayList<>());
    }

    public void clear() {
        messagesByTopic.clear();
    }
}
```

Version Highlights

Spring Boot Kafka Integration Evolution

Version	Release	Key Features
Spring Boot 3.2.x	2024	**Enhanced auto-configuration**, improved observability
Spring Boot 3.1.x	2023	**Native compilation** support, GraalVM compatibility
Spring Boot 3.0.x	2022	**Spring Framework 6**, Jakarta EE migration
Spring Boot 2.7.x	2022	**Kafka 3.x support**, enhanced health indicators
Spring Boot 2.6.x	2021	**@EmbeddedKafka improvements**, better testing support

Version	Release	Key Features
Spring Boot 2.5.x	2021	**Kafka Streams auto-configuration**, metrics enhancements
Spring Boot 2.4.x	2020	**Configuration properties**, validation improvements
Spring Boot 2.3.x	2020	**Docker support**, embedded Kafka enhancements
Spring Boot 2.2.x	2019	**Kafka 2.x support**, transaction improvements
Spring Boot 2.1.x	2018	**Initial comprehensive** Kafka auto-configuration

Spring Kafka Version Compatibility

Spring Kafka 3.2.x (2024):

- **Spring Boot 3.2+ compatibility**
- **Enhanced observability** with Micrometer
- **Improved error handling** and retry mechanisms
- **Better testing utilities** for embedded Kafka

Spring Kafka 3.1.x (2023):

- **Native compilation** support for GraalVM
- **Improved batch processing** performance
- **Enhanced security** configuration options

Spring Kafka 3.0.x (2022):

- **Spring Framework 6** compatibility
- **Jakarta EE namespace** migration
- **Improved KRaft support**

Spring Kafka 2.9.x (2022):

- **Kafka 3.x broker** compatibility
- **Enhanced metrics** and monitoring
- **Improved testing** annotations

Key Milestones Timeline

```
Spring Boot Kafka Integration Timeline:

2018 ████████████████████████████████████████ Spring Boot 2.1
     Initial comprehensive auto-configuration

2019 ████████████████████████████████████████ Spring Boot 2.2
     Kafka 2.x support, transaction improvements

2020 ████████████████████████████████████████ Spring Boot 2.3
     Docker support, @EmbeddedKafka enhancements
```

```
2020 ████████████████████████████████████ Spring Boot 2.4
     Configuration properties validation

2021 ████████████████████████████████████ Spring Boot 2.5
     Kafka Streams auto-configuration

2021 ████████████████████████████████████ Spring Boot 2.6
     Testing improvements, @EmbeddedKafka fixes

2022 ████████████████████████████████████ Spring Boot 2.7
     Kafka 3.x support, enhanced health checks

2022 ████████████████████████████████████ Spring Boot 3.0
     Jakarta EE migration, Spring Framework 6

2023 ████████████████████████████████████ Spring Boot 3.1
     Native compilation, GraalVM support

2024 ████████████████████████████████████ Spring Boot 3.2
     Enhanced auto-configuration, observability

Key Features by Era:
├─ 2018-2019: Foundation & Core Features
├─ 2020-2021: Testing & Configuration
├─ 2022-2023: Modernization & Migration
└─ 2024-2025: Performance & Observability
```

Key Takeaways

Essential Integration Patterns

1. **Auto-Configuration**: Leverage Spring Boot's convention-over-configuration for rapid development
2. **Property-Based Setup**: Use application.yml/properties for environment-specific configuration
3. **Embedded Testing**: Employ @EmbeddedKafka for fast, reliable integration tests
4. **Production Monitoring**: Implement health checks, metrics, and error handling
5. **Gradual Customization**: Start with auto-configuration, customize only when needed

Configuration Best Practices

- **Use Property Profiles**: Different configurations for dev, test, prod
- **Enable Health Checks**: Monitor Kafka connectivity and performance
- **Implement Error Handling**: Dead letter topics and retry mechanisms
- **Test Thoroughly**: Comprehensive integration tests with @EmbeddedKafka
- **Monitor Metrics**: Track message flow and system health

Performance Considerations

- **Auto-configuration has minimal overhead** (~2-5% performance impact)
- **Property-based configuration provides optimal balance** of simplicity and performance
- **@EmbeddedKafka adds ~500ms to test startup** but enables fast, isolated testing

- **Batch processing and compression** significantly improve throughput
- **Connection pooling and proper sizing** are crucial for production

Last Updated: September 2025
Spring Boot Version Coverage: 3.2.x
Spring Kafka Version: 3.2.x
Apache Kafka Compatibility: 3.6.x

This comprehensive Spring Boot Kafka Integration guide provides production-ready patterns for implementing Kafka applications with Spring Boot, from basic auto-configuration to advanced testing strategies, ensuring rapid development without sacrificing reliability or performance.

[650] [651]