Spring Kafka Advanced Features: Part 3 - Multi-tenancy & Best Practices

Final part of the comprehensive Spring Kafka Advanced Features guide covering multi-tenancy architectures, performance comparisons, best practices, common pitfalls, and version highlights for production deployment.

🏢 Multi-tenancy Setups

Simple Explanation: Multi-tenancy in Spring Kafka enables a single application to serve multiple tenants (customers, organizations, or environments) with isolated data streams, security boundaries, and configuration management. It provides tenant-aware routing, topic isolation, and resource management while maintaining operational efficiency.

What Problem It Solves:

- **Tenant Isolation**: Separate data and processing for different customers/organizations
- **Scalable Architecture**: Single application serving multiple tenants efficiently
- **Security Boundaries**: Ensure tenant data privacy and access control
- **Configuration Management**: Tenant-specific configurations and behavior
- **Cost Optimization**: Shared infrastructure with tenant-specific billing/monitoring

Multi-tenancy Architecture:

```
Spring Kafka Multi-Tenancy Architecture:

┌─────────────────────────────────────────────────────────────────┐
│                Spring Boot Application                          │
│                                                                 │
│  ┌─────────────────────────────────────────────────────────┐   │
│  │            Multi-Tenant Request Layer                   │   │
│  │                                                         │   │
│  │  HTTP Headers: X-Tenant-ID: tenant-a                    │   │
│  │  JWT Claims: tenant_id, tenant_permissions              │   │
│  │  URL Paths: /api/tenant-a/orders                        │   │
│  └─────────────────────────────────────────────────────────┘   │
│            │                                                    │
│            ▼                                                    │
│  ┌─────────────────────────────────────────────────────────┐   │
│  │           Tenant Context & Routing                      │   │
│  │                                                         │   │
│  │  ┌──────────────┐ ┌──────────────┐ ┌──────────────┐     │   │
│  │  │  Tenant A    │ │  Tenant B    │ │  Tenant C    │     │   │
│  │  │ Context      │ │ Context      │ │ Context      │     │   │
│  │  └──────────────┘ └──────────────┘ └──────────────┘     │   │
│  └─────────────────────────────────────────────────────────┘   │
│            │                                                    │
│            ▼                                                    │
│  ┌─────────────────────────────────────────────────────────┐   │
```

```
│  │          Tenant-Aware Kafka Operations                  │ │
│  │                                                         │ │
│  │  ┌──────────────────────────────────────────────────┐ │ │
│  │  │        Dynamic Topic Routing                      │ │ │
│  │  │  tenant-a-orders, tenant-a-customers              │ │ │
│  │  │  tenant-b-orders, tenant-b-customers              │ │ │
│  │  │  tenant-c-orders, tenant-c-customers              │ │ │
│  │  └──────────────────────────────────────────────────┘ │ │
│  │                                                         │ │
│  │  ┌──────────────────────────────────────────────────┐ │ │
│  │  │      Tenant-Specific Configuration                │ │ │
│  │  │  • Custom serializers per tenant                  │ │ │
│  │  │  • Different retention policies                   │ │ │
│  │  │  • Tenant-specific security settings              │ │ │
│  │  └──────────────────────────────────────────────────┘ │ │
│  └──────────────────────────────────────────────────────┘ │
└──────────────────────────────────────────────────────────┘
                        │
                        │ Tenant-isolated messages
                        ▼
┌──────────────────────────────────────────────────────────┐
│                   Kafka Cluster                           │
│                                                           │
│  ┌──────────────────────────────────────────────────────┐ │
│  │                Topic Isolation                        │ │
│  │                                                       │ │
│  │  tenant-a-orders-0  │  tenant-b-orders-0  │ tenant-c... │ │
│  │  tenant-a-orders-1  │  tenant-b-orders-1  │ ...         │ │
│  │  tenant-a-orders-2  │  tenant-b-orders-2  │ ...         │ │
│  │                                                       │ │
│  │  tenant-a-events-0  │  tenant-b-events-0  │ tenant-c... │ │
│  │  tenant-a-events-1  │  tenant-b-events-1  │ ...         │ │
│  └──────────────────────────────────────────────────────┘ │
│                                                           │
│  ┌──────────────────────────────────────────────────────┐ │
│  │                ACL Security                           │ │
│  │                                                       │ │
│  │  tenant-a-user → tenant-a-* topics (READ/WRITE)       │ │
│  │  tenant-b-user → tenant-b-* topics (READ/WRITE)       │ │
│  │  tenant-c-user → tenant-c-* topics (READ/WRITE)       │ │
│  └──────────────────────────────────────────────────────┘ │
└──────────────────────────────────────────────────────────┘

Multi-Tenancy Strategies:
1. Topic-level isolation: tenant-{id}-{topic-name}
2. Partition-level isolation: hash(tenant-id) → partition
3. Cluster-level isolation: separate Kafka clusters per tenant
4. Application-level filtering: shared topics with tenant filtering
```

Complete Multi-tenancy Implementation

```java
/**
 * Tenant context management
 */
@Component
@lombok.extern.slf4j.Slf4j
public class TenantContextHolder {

    private static final ThreadLocal<String> tenantContext = new ThreadLocal<>();

    public static void setTenantId(String tenantId) {
        if (tenantId == null || tenantId.trim().isEmpty()) {
            throw new IllegalArgumentException("Tenant ID cannot be null or
empty");
        }

        tenantContext.set(tenantId.trim().toLowerCase());
        log.debug("Set tenant context: {}", tenantId);
    }

    public static String getTenantId() {
        String tenantId = tenantContext.get();
        if (tenantId == null) {
            throw new IllegalStateException("No tenant context found");
        }
        return tenantId;
    }

    public static Optional<String> getTenantIdOptional() {
        return Optional.ofNullable(tenantContext.get());
    }

    public static void clear() {
        tenantContext.remove();
        log.debug("Cleared tenant context");
    }

    public static boolean hasTenantContext() {
        return tenantContext.get() != null;
    }
}

/**
 * Tenant extraction from various sources
 */
@Component
@lombok.extern.slf4j.Slf4j
public class TenantResolver {

    private final List<TenantExtractionStrategy> extractionStrategies;

    public TenantResolver() {
        this.extractionStrategies = Arrays.asList(
            new HeaderTenantExtractionStrategy(),
```

```
            new JwtTenantExtractionStrategy(),
            new PathVariableTenantExtractionStrategy(),
            new QueryParameterTenantExtractionStrategy()
        );
    }

    public Optional<String> resolveTenant(HttpServletRequest request) {

        for (TenantExtractionStrategy strategy : extractionStrategies) {
            Optional<String> tenantId = strategy.extractTenant(request);
            if (tenantId.isPresent()) {
                log.debug("Resolved tenant {} using strategy: {}",
                    tenantId.get(), strategy.getClass().getSimpleName());
                return tenantId;
            }
        }

        log.warn("No tenant could be resolved from request: {}",
request.getRequestURI());
        return Optional.empty();
    }

    // Tenant extraction strategies
    interface TenantExtractionStrategy {
        Optional<String> extractTenant(HttpServletRequest request);
    }

    static class HeaderTenantExtractionStrategy implements
TenantExtractionStrategy {
        @Override
        public Optional<String> extractTenant(HttpServletRequest request) {
            String tenantId = request.getHeader("X-Tenant-ID");
            return Optional.ofNullable(tenantId);
        }
    }

    static class JwtTenantExtractionStrategy implements TenantExtractionStrategy {
        @Override
        public Optional<String> extractTenant(HttpServletRequest request) {
            String authHeader = request.getHeader("Authorization");
            if (authHeader != null && authHeader.startsWith("Bearer ")) {
                try {
                    String token = authHeader.substring(7);
                    // Simplified JWT parsing - use proper JWT library in
production
                    String[] parts = token.split("\\.");
                    if (parts.length == 3) {
                        String payload = new
String(Base64.getDecoder().decode(parts[1]));
                        // Extract tenant from JWT claims
                        return extractTenantFromJwtPayload(payload);
                    }
                } catch (Exception e) {
                    // Log but don't fail, try other strategies
```

```
                }
            }
            return Optional.empty();
        }

        private Optional<String> extractTenantFromJwtPayload(String payload) {
            try {
                ObjectMapper mapper = new ObjectMapper();
                JsonNode claims = mapper.readTree(payload);
                JsonNode tenantNode = claims.get("tenant_id");
                return tenantNode != null ? Optional.of(tenantNode.asText()) :
Optional.empty();
            } catch (Exception e) {
                return Optional.empty();
            }
        }
    }

    static class PathVariableTenantExtractionStrategy implements
TenantExtractionStrategy {
        @Override
        public Optional<String> extractTenant(HttpServletRequest request) {
            String path = request.getRequestURI();
            // Pattern: /api/{tenantId}/...
            String[] segments = path.split("/");
            if (segments.length >= 3 && "api".equals(segments[1])) {
                return Optional.of(segments[2]);
            }
            return Optional.empty();
        }
    }

    static class QueryParameterTenantExtractionStrategy implements
TenantExtractionStrategy {
        @Override
        public Optional<String> extractTenant(HttpServletRequest request) {
            String tenantId = request.getParameter("tenantId");
            return Optional.ofNullable(tenantId);
        }
    }
}

/**
 * Multi-tenant interceptor for web requests
 */
@Component
@lombok.extern.slf4j.Slf4j
public class MultiTenantInterceptor implements HandlerInterceptor {

    @Autowired
    private TenantResolver tenantResolver;

    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse
```

```
response, Object handler) {

        try {
            Optional<String> tenantId = tenantResolver.resolveTenant(request);

            if (tenantId.isPresent()) {
                TenantContextHolder.setTenantId(tenantId.get());

                // Add tenant ID to MDC for logging
                MDC.put("tenantId", tenantId.get());

                log.debug("Set tenant context for request: {} -> {}",
                    request.getRequestURI(), tenantId.get());

                return true;

            } else {
                log.warn("No tenant context found for request: {}",
request.getRequestURI());

                // Decide whether to allow requests without tenant context
                if (requiresTenantContext(request)) {
                    response.setStatus(HttpStatus.BAD_REQUEST.value());
                    response.getWriter().write("{\"error\": \"Tenant context
required\"}");
                    return false;
                }

                return true;
            }

        } catch (Exception e) {
            log.error("Error setting tenant context", e);
            return false;
        }
    }

    @Override
    public void afterCompletion(HttpServletRequest request, HttpServletResponse
response,
                                Object handler, Exception ex) {

        // Clean up tenant context
        TenantContextHolder.clear();
        MDC.remove("tenantId");
    }

    private boolean requiresTenantContext(HttpServletRequest request) {
        String path = request.getRequestURI();

        // Skip tenant requirement for health checks and public endpoints
        return !path.startsWith("/actuator") &&
               !path.startsWith("/public") &&
               !path.equals("/") &&
```

```
                !path.equals("/health");
    }
}

/**
 * Multi-tenant Kafka configuration
 */
@Configuration
@EnableKafka
@lombok.extern.slf4j.Slf4j
public class MultiTenantKafkaConfiguration {

    @Value("${kafka.multi-tenant.topic-prefix:}")
    private String topicPrefix;

    @Value("${kafka.multi-tenant.default-tenant:default}")
    private String defaultTenant;

    /**
     * Multi-tenant topic name resolver
     */
    @Bean
    public MultiTenantTopicResolver multiTenantTopicResolver() {
        return new MultiTenantTopicResolver(topicPrefix);
    }

    /**
     * Tenant-aware KafkaTemplate
     */
    @Bean
    @Primary
    public KafkaTemplate<String, Object>
multiTenantKafkaTemplate(ProducerFactory<String, Object> producerFactory) {

        KafkaTemplate<String, Object> template = new KafkaTemplate<>
(producerFactory) {
            @Override
            public ListenableFuture<SendResult<String, Object>> send(String topic,
String key, Object data) {
                String tenantAwareTopic = resolveTenantTopic(topic);
                return super.send(tenantAwareTopic, key, data);
            }

            @Override
            public ListenableFuture<SendResult<String, Object>>
send(ProducerRecord<String, Object> record) {
                String tenantAwareTopic = resolveTenantTopic(record.topic());
                ProducerRecord<String, Object> tenantRecord = new ProducerRecord<>
(
                    tenantAwareTopic,
                    record.partition(),
                    record.timestamp(),
                    record.key(),
                    record.value(),
```

```
                    record.headers()
                );
                return super.send(tenantRecord);
            }
        };

        // Add tenant information to headers
        template.setProducerInterceptors(Collections.singletonList(new
TenantProducerInterceptor()));

        log.info("Configured multi-tenant KafkaTemplate");

        return template;
    }

    private String resolveTenantTopic(String baseTopic) {
        try {
            String tenantId = TenantContextHolder.getTenantId();
            return String.format("%s%s-%s",
                topicPrefix.isEmpty() ? "" : topicPrefix + "-",
                tenantId,
                baseTopic);
        } catch (IllegalStateException e) {
            // No tenant context, use default
            log.debug("No tenant context, using default tenant for topic: {}",
baseTopic);
            return String.format("%s%s-%s",
                topicPrefix.isEmpty() ? "" : topicPrefix + "-",
                defaultTenant,
                baseTopic);
        }
    }

    /**
     * Multi-tenant listener container factory
     */
    @Bean
    @Primary
    public ConcurrentKafkaListenerContainerFactory<String, Object>
multiTenantKafkaListenerContainerFactory(
            ConsumerFactory<String, Object> consumerFactory) {

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

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

        // Add tenant extraction from consumer records
        factory.setConsumerInterceptors(Collections.singletonList(new
TenantConsumerInterceptor()));

        log.info("Configured multi-tenant KafkaListenerContainerFactory");
```

```java
        return factory;
    }
}

/**
 * Multi-tenant topic name resolver
 */
@Component
public class MultiTenantTopicResolver {

    private final String topicPrefix;

    public MultiTenantTopicResolver(String topicPrefix) {
        this.topicPrefix = topicPrefix != null ? topicPrefix : "";
    }

    public String resolveTopicName(String baseTopic, String tenantId) {
        if (tenantId == null || tenantId.trim().isEmpty()) {
            throw new IllegalArgumentException("Tenant ID cannot be null or
empty");
        }

        return String.format("%s%s-%s",
            topicPrefix.isEmpty() ? "" : topicPrefix + "-",
            tenantId.toLowerCase(),
            baseTopic);
    }

    public String resolveTopicName(String baseTopic) {
        String tenantId = TenantContextHolder.getTenantId();
        return resolveTopicName(baseTopic, tenantId);
    }

    public Optional<String> extractTenantFromTopic(String topicName) {
        String pattern = topicPrefix.isEmpty() ?
            "^([a-zA-Z0-9-]+)-(.+)$" :
            "^" + Pattern.quote(topicPrefix) + "-([a-zA-Z0-9-]+)-(.+)$";

        Pattern regex = Pattern.compile(pattern);
        Matcher matcher = regex.matcher(topicName);

        if (matcher.matches()) {
            return Optional.of(matcher.group(1));
        }

        return Optional.empty();
    }

    public String extractBaseTopicFromTenantTopic(String tenantTopic) {
        String pattern = topicPrefix.isEmpty() ?
            "^[a-zA-Z0-9-]+-(.+)$" :
            "^" + Pattern.quote(topicPrefix) + "-[a-zA-Z0-9-]+-(.+)$";

        Pattern regex = Pattern.compile(pattern);
```

```
        Matcher matcher = regex.matcher(tenantTopic);

        if (matcher.matches()) {
            return matcher.group(1);
        }

        return tenantTopic; // Return as-is if pattern doesn't match
    }
}

/**
 * Producer interceptor to add tenant information
 */
public class TenantProducerInterceptor implements ProducerInterceptor<String,
Object> {

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

        // Add tenant ID to headers if available
        try {
            String tenantId = TenantContextHolder.getTenantId();
            record.headers().add("tenant-id",
tenantId.getBytes(StandardCharsets.UTF_8));
            record.headers().add("tenant-timestamp",

String.valueOf(System.currentTimeMillis()).getBytes(StandardCharsets.UTF_8));
        } catch (IllegalStateException e) {
            // No tenant context available
        }

        return record;
    }

    @Override
    public void onAcknowledgement(RecordMetadata metadata, Exception exception) {
        // Could add tenant-specific metrics here
    }

    @Override
    public void close() {}

    @Override
    public void configure(Map<String, ?> configs) {}
}

/**
 * Consumer interceptor to extract tenant information
 */
public class TenantConsumerInterceptor implements ConsumerInterceptor<String,
Object> {

    @Override
```

```java
    public ConsumerRecords<String, Object> onConsume(ConsumerRecords<String,
Object> records) {

        for (ConsumerRecord<String, Object> record : records) {
            // Extract tenant ID from headers or topic name
            Header tenantHeader = record.headers().lastHeader("tenant-id");
            String tenantId = null;

            if (tenantHeader != null) {
                tenantId = new String(tenantHeader.value(),
StandardCharsets.UTF_8);
            } else {
                // Try to extract from topic name
                Optional<String> extracted =
extractTenantFromTopic(record.topic());
                if (extracted.isPresent()) {
                    tenantId = extracted.get();
                }
            }

            if (tenantId != null) {
                // Add tenant ID back to headers for processing
                record.headers().add("extracted-tenant-id",
tenantId.getBytes(StandardCharsets.UTF_8));
            }
        }

        return records;
    }

    @Override
    public void onCommit(Map<TopicPartition, OffsetAndMetadata> offsets) {
        // Could add tenant-specific commit metrics here
    }

    @Override
    public void close() {}

    @Override
    public void configure(Map<String, ?> configs) {}

    private Optional<String> extractTenantFromTopic(String topicName) {
        // Simple pattern: tenant-id-base-topic
        String[] parts = topicName.split("-", 3);
        if (parts.length >= 2) {
            return Optional.of(parts[0]);
        }
        return Optional.empty();
    }
}

/**
 * Multi-tenant service implementation
 */
```

```java
@Service
@lombok.extern.slf4j.Slf4j
public class MultiTenantKafkaService {

    @Autowired
    private KafkaTemplate<String, Object> kafkaTemplate;

    @Autowired
    private MultiTenantTopicResolver topicResolver;

    /**
     * Send message with tenant awareness
     */
    public void sendTenantMessage(String baseTopic, String key, Object message) {

        String tenantId = TenantContextHolder.getTenantId();
        String tenantTopic = topicResolver.resolveTopicName(baseTopic, tenantId);

        log.info("Sending message to tenant topic: {} (tenant: {}, base: {})",
            tenantTopic, tenantId, baseTopic);

        // Add tenant context to message if it's a map
        if (message instanceof Map) {
            @SuppressWarnings("unchecked")
            Map<String, Object> messageMap = (Map<String, Object>) message;
            messageMap.put("tenantId", tenantId);
            messageMap.put("tenantTimestamp", Instant.now());
        }

        kafkaTemplate.send(tenantTopic, key, message);
    }

    /**
     * Send message to specific tenant
     */
    public void sendMessageToTenant(String tenantId, String baseTopic, String key,
Object message) {

        String currentTenant =
TenantContextHolder.getTenantIdOptional().orElse(null);

        try {
            // Temporarily set tenant context
            TenantContextHolder.setTenantId(tenantId);

            String tenantTopic = topicResolver.resolveTopicName(baseTopic,
tenantId);

            log.info("Sending message to specific tenant: {} -> {}", tenantId,
tenantTopic);

            kafkaTemplate.send(tenantTopic, key, message);

        } finally {
```

```
            // Restore original tenant context
            if (currentTenant != null) {
                TenantContextHolder.setTenantId(currentTenant);
            } else {
                TenantContextHolder.clear();
            }
        }
    }

    /**
     * Broadcast message to all tenants
     */
    public void broadcastMessageToAllTenants(List<String> tenantIds, String
baseTopic, String key, Object message) {

        String currentTenant =
TenantContextHolder.getTenantIdOptional().orElse(null);

        for (String tenantId : tenantIds) {
            try {
                TenantContextHolder.setTenantId(tenantId);

                String tenantTopic = topicResolver.resolveTopicName(baseTopic,
tenantId);

                // Create tenant-specific message
                Map<String, Object> tenantMessage = new HashMap<>();
                tenantMessage.put("originalMessage", message);
                tenantMessage.put("targetTenant", tenantId);
                tenantMessage.put("broadcastTimestamp", Instant.now());

                kafkaTemplate.send(tenantTopic, key, tenantMessage);

                log.debug("Broadcasted message to tenant: {}", tenantId);

            } catch (Exception e) {
                log.error("Failed to broadcast message to tenant: {}", tenantId,
e);
            }
        }

        // Restore original tenant context
        if (currentTenant != null) {
            TenantContextHolder.setTenantId(currentTenant);
        } else {
            TenantContextHolder.clear();
        }

        log.info("Broadcasted message to {} tenants", tenantIds.size());
    }
}

/**
 * Multi-tenant message consumer
```

```
 */
@Component
@lombok.extern.slf4j.Slf4j
public class MultiTenantMessageConsumer {

    @Autowired
    private MultiTenantTopicResolver topicResolver;

    /**
     * Generic tenant-aware message listener
     */
    @KafkaListener(
        topics = "#{multiTenantTopicResolver.resolveTopicName('user-events')}",
        groupId = "multi-tenant-user-processor",
        containerFactory = "multiTenantKafkaListenerContainerFactory"
    )
    public void handleUserEvents(@Payload UserEvent userEvent,
                                 @Header(KafkaHeaders.RECEIVED_TOPIC) String topic,
                                 @Header(name = "tenant-id", required = false)
String headerTenantId,
                                 @Header(name = "extracted-tenant-id", required =
false) String extractedTenantId) {

        // Determine tenant ID from various sources
        String tenantId = headerTenantId != null ? headerTenantId :
extractedTenantId;

        if (tenantId == null) {
            tenantId =
topicResolver.extractTenantFromTopic(topic).orElse("unknown");
        }

        // Set tenant context for processing
        TenantContextHolder.setTenantId(tenantId);

        try {
            log.info("Processing user event for tenant: {} -> {}", tenantId,
userEvent.getUserId());

            // Add tenant-specific processing logic here
            processUserEventForTenant(tenantId, userEvent);

        } finally {
            TenantContextHolder.clear();
        }
    }

    /**
     * Tenant-specific processing method
     */
    private void processUserEventForTenant(String tenantId, UserEvent userEvent) {

        // Load tenant-specific configuration or rules
        TenantConfiguration tenantConfig = loadTenantConfiguration(tenantId);
```

```
        if (tenantConfig.isProcessingEnabled()) {

            // Apply tenant-specific business rules
            if (tenantConfig.getMaxUserAge() != null &&
                userEvent.getUserAge() != null &&
                userEvent.getUserAge() > tenantConfig.getMaxUserAge()) {

                log.warn("User age exceeds tenant limit: {} > {} for tenant {}",
                    userEvent.getUserAge(), tenantConfig.getMaxUserAge(),
tenantId);
                return;
            }

            // Process based on tenant tier
            switch (tenantConfig.getTier()) {
                case "PREMIUM":
                    processPremiumUserEvent(tenantId, userEvent);
                    break;
                case "STANDARD":
                    processStandardUserEvent(tenantId, userEvent);
                    break;
                case "BASIC":
                    processBasicUserEvent(tenantId, userEvent);
                    break;
                default:
                    log.warn("Unknown tenant tier: {} for tenant {}",
tenantConfig.getTier(), tenantId);
            }

        } else {
            log.info("Processing disabled for tenant: {}", tenantId);
        }
    }

    private TenantConfiguration loadTenantConfiguration(String tenantId) {
        // In production, load from database or configuration service
        return TenantConfiguration.builder()
            .tenantId(tenantId)
            .tier("STANDARD")
            .processingEnabled(true)
            .maxUserAge(120)
            .build();
    }

    private void processPremiumUserEvent(String tenantId, UserEvent event) {
        log.info("Processing premium user event for tenant: {}", tenantId);
        // Premium-specific processing
    }

    private void processStandardUserEvent(String tenantId, UserEvent event) {
        log.info("Processing standard user event for tenant: {}", tenantId);
        // Standard processing
    }
```

```java
    private void processBasicUserEvent(String tenantId, UserEvent event) {
        log.info("Processing basic user event for tenant: {}", tenantId);
        // Basic processing
    }
}

/**
 * Multi-tenant REST controller
 */
@RestController
@RequestMapping("/api/{tenantId}")
@lombok.extern.slf4j.Slf4j
public class MultiTenantController {

    @Autowired
    private MultiTenantKafkaService kafkaService;

    @PostMapping("/events")
    public ResponseEntity<Map<String, String>> createEvent(@PathVariable String
tenantId,
                                                            @RequestBody Map<String,
Object> eventData) {

        try {
            // Tenant context should already be set by interceptor
            log.info("Creating event for tenant: {}", tenantId);

            // Validate tenant context matches path variable
            String contextTenantId = TenantContextHolder.getTenantId();
            if (!tenantId.equals(contextTenantId)) {
                return ResponseEntity.badRequest()
                    .body(Map.of("error", "Tenant context mismatch"));
            }

            // Add metadata
            eventData.put("eventId", UUID.randomUUID().toString());
            eventData.put("timestamp", Instant.now().toString());

            // Send to tenant-specific topic
            kafkaService.sendTenantMessage("events", (String)
eventData.get("eventId"), eventData);

            return ResponseEntity.ok(Map.of(
                "status", "ACCEPTED",
                "tenantId", tenantId,
                "eventId", (String) eventData.get("eventId")
            ));

        } catch (Exception e) {
            log.error("Error creating event for tenant: {}", tenantId, e);

            return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR)
                .body(Map.of("error", "Failed to create event"));
```

```
        }
    }

    @PostMapping("/broadcast")
    public ResponseEntity<Map<String, String>> broadcastEvent(@PathVariable String
tenantId,
                                                              @RequestBody
BroadcastRequest request) {

        try {
            log.info("Broadcasting event from tenant: {} to {} targets",
                tenantId, request.getTargetTenants().size());

            Map<String, Object> eventData = new HashMap<>(request.getEventData());
            eventData.put("broadcastId", UUID.randomUUID().toString());
            eventData.put("sourceTenant", tenantId);

            kafkaService.broadcastMessageToAllTenants(
                request.getTargetTenants(),
                "broadcast-events",
                (String) eventData.get("broadcastId"),
                eventData
            );

            return ResponseEntity.ok(Map.of(
                "status", "BROADCASTED",
                "sourceTenant", tenantId,
                "targetCount", String.valueOf(request.getTargetTenants().size())
            ));

        } catch (Exception e) {
            log.error("Error broadcasting event from tenant: {}", tenantId, e);

            return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR)
                .body(Map.of("error", "Failed to broadcast event"));
        }
    }
}

// Supporting data classes for multi-tenancy
@lombok.Data
@lombok.Builder
@lombok.NoArgsConstructor
@lombok.AllArgsConstructor
class TenantConfiguration {
    private String tenantId;
    private String tier;
    private boolean processingEnabled;
    private Integer maxUserAge;
    private Map<String, Object> customSettings;
}

@lombok.Data
@lombok.AllArgsConstructor
```

```java
@lombok.NoArgsConstructor
class BroadcastRequest {
    private List<String> targetTenants;
    private Map<String, Object> eventData;
}

@lombok.Data
@lombok.Builder
@lombok.NoArgsConstructor
@lombok.AllArgsConstructor
class UserEvent {
    private String userId;
    private String eventType;
    private Integer userAge;
    private Map<String, Object> eventData;
    private Instant timestamp;
}
```

📊 Comparisons & Trade-offs

Advanced Features Comparison

Feature	Complexity	Performance Impact	Use Cases	Best For
ReplyingKafkaTemplate	⭐⭐⭐ Medium	⭐⭐⭐ Medium	Sync communication, API gateways	**Request/response patterns**
Kafka Streams	⭐⭐⭐⭐ High	⭐⭐ High	Real-time analytics, CEP	**Stream processing**
Kafka Connect	⭐⭐⭐⭐ High	⭐⭐⭐ Medium	Data pipelines, ETL	**System integration**
Multi-tenancy	⭐⭐⭐⭐⭐ Very High	⭐⭐⭐ Medium	SaaS platforms, enterprises	**Multi-customer systems**

Request/Reply vs Message Queuing

Aspect	Request/Reply	Traditional Messaging
Latency	Higher (sync wait)	Lower (async)
Throughput	Lower (blocking)	Higher (non-blocking)
Error Handling	Immediate feedback	Eventual consistency
Complexity	Medium (correlation)	Low (fire-and-forget)
Use Case	API integration	Event streaming

Kafka Streams vs Traditional Processing

Aspect	Kafka Streams	Batch Processing	Traditional CEP
Latency	Near real-time	High (batch intervals)	Real-time
Scalability	Excellent	Good	Limited
State Management	Built-in	External	External
Fault Tolerance	Automatic	Manual	Manual
Learning Curve	Steep	Medium	Steep

🚨 Common Pitfalls & Best Practices

Critical Anti-Patterns

❌ Request/Reply Mistakes

```
// DON'T - Blocking without timeout
public class BadRequestReplyService {

    public String sendRequest(String message) {
        try {
            // BAD: No timeout - can hang forever
            RequestReplyFuture<String, String, String> future =
                replyingTemplate.sendAndReceive(new ProducerRecord<>("requests",
message));

            // This can block indefinitely!
            ConsumerRecord<String, String> response = future.get();
            return response.value();

        } catch (Exception e) {
            // BAD: Not handling specific exceptions
            throw new RuntimeException(e);
        }
    }

    // BAD: Creating new templates for each request
    public String anotherBadPattern(String message) {
        ReplyingKafkaTemplate<String, String, String> template =
            new ReplyingKafkaTemplate<>(producerFactory, replyContainer);
        // This is inefficient and can cause resource leaks
        return "response";
    }
}
```

❌ Kafka Streams Mistakes

```java
// DON'T - Improper state store usage and resource management
public class BadStreamsConfiguration {

    @Bean
    public KStream<String, String> badStreamTopology(StreamsBuilder builder) {

        // BAD: No error handling in stream processing
        return builder.stream("input-topic")
            .mapValues(value -> {
                // This can throw exceptions and crash the stream
                return processValue(value); // No try-catch!
            })
            .filter((key, value) -> value != null); // Filter should be before
processing
    }

    // BAD: Creating new Serdes for each operation
    @Bean
    public KTable<String, Long> badAggregation(StreamsBuilder builder) {

        KStream<String, String> stream = builder.stream("events");

        return stream
            .groupByKey()
            .count(Materialized.<String, Long, KeyValueStore<Bytes,
byte[]>>as("bad-store")
                .withKeySerde(Serdes.String()) // Creating new Serde each time
                .withValueSerde(Serdes.Long())); // Memory leak potential
    }
}
```

❌ Multi-tenancy Mistakes

```java
// DON'T - Security vulnerabilities and context leaks
public class BadMultiTenantService {

    // BAD: No tenant validation - security vulnerability
    public void sendMessageToTenant(String tenantId, String message) {
        // No validation if user has access to this tenant!
        String topic = tenantId + "-events";
        kafkaTemplate.send(topic, message);
    }

    // BAD: Tenant context leaking between requests
    private static String currentTenant; // Static variable - thread unsafe!

    public void processMessage(String message) {
        // BAD: Not clearing context can cause cross-tenant data leaks
        currentTenant = extractTenant(message);
```

```
        // Process without clearing context
        businessLogic(message);
        // Context never cleared - will affect next request!
    }
}
```

Production Best Practices

☑ Request/Reply Excellence

```
/**
 * ☑ GOOD - Production-ready request/reply implementation
 */
@Service
@lombok.extern.slf4j.Slf4j
public class ProductionRequestReplyService {

    @Autowired
    private ReplyingKafkaTemplate<String, Object, Object> replyingTemplate;

    @Value("${request-reply.timeout.seconds:30}")
    private Integer timeoutSeconds;

    @Value("${request-reply.retry.attempts:3}")
    private Integer retryAttempts;

    /**
     * ☑ GOOD - Proper timeout, error handling, and retry logic
     */
    @Retryable(
        value = {TimeoutException.class, ExecutionException.class},
        maxAttempts = 3,
        backoff = @Backoff(delay = 1000, multiplier = 2)
    )
    public <T> Optional<T> sendRequestSafely(String topic, String key, Object
request, Class<T> responseType) {

        ProducerRecord<String, Object> record = new ProducerRecord<>(topic, key,
request);

        // Add request metadata
        record.headers().add("request-id",
UUID.randomUUID().toString().getBytes());
        record.headers().add("request-timestamp",
String.valueOf(System.currentTimeMillis()).getBytes());
        record.headers().add("client-version", "1.0".getBytes());

        Timer.Sample sample = Timer.start(Metrics.globalRegistry);

        try {
            log.info("Sending request: topic={}, key={}, type={}", topic, key,
```

```
request.getClass().getSimpleName());

            RequestReplyFuture<String, Object, Object> future =
                replyingTemplate.sendAndReceive(record,
Duration.ofSeconds(timeoutSeconds));

            // Wait for response with timeout
            ConsumerRecord<String, Object> response = future.get(timeoutSeconds,
TimeUnit.SECONDS);

            log.info("Received response: topic={}, key={}, partition={}, offset=
{}",
                response.topic(), response.key(), response.partition(),
response.offset());

            // Convert response safely
            Object responseValue = response.value();
            if (responseType.isInstance(responseValue)) {

                sample.stop(Timer.builder("request.reply.success")
                    .tag("topic", topic)
                    .register(Metrics.globalRegistry));

                return Optional.of(responseType.cast(responseValue));

            } else {
                log.warn("Response type mismatch: expected {}, got {}",
                    responseType.getName(), responseValue.getClass().getName());

                return Optional.empty();
            }

        } catch (TimeoutException e) {

            sample.stop(Timer.builder("request.reply.timeout")
                .tag("topic", topic)
                .register(Metrics.globalRegistry));

            log.error("Request timeout: topic={}, key={}, timeout={}s", topic,
key, timeoutSeconds);
            throw e;

        } catch (ExecutionException e) {

            sample.stop(Timer.builder("request.reply.error")
                .tag("topic", topic)
                .tag("error.type", e.getCause().getClass().getSimpleName())
                .register(Metrics.globalRegistry));

            log.error("Request execution failed: topic={}, key={}", topic, key,
e);
            throw e;

        } catch (InterruptedException e) {
```

```
            Thread.currentThread().interrupt();

            sample.stop(Timer.builder("request.reply.interrupted")
                .tag("topic", topic)
                .register(Metrics.globalRegistry));

            log.error("Request interrupted: topic={}, key={}", topic, key);
            return Optional.empty();
        }
    }
}
```

☑ Kafka Streams Excellence

```
/**
 * ☑ GOOD - Production-ready Kafka Streams configuration
 */
@Configuration
@EnableKafkaStreams
@lombok.extern.slf4j.Slf4j
public class ProductionStreamsConfiguration {

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

        // Define Serdes once and reuse
        Serde<OrderEvent> orderSerde = Serdes.serdeFrom(new JsonSerializer<>(),
new JsonDeserializer<>(OrderEvent.class));
        Serde<EnrichedOrder> enrichedSerde = Serdes.serdeFrom(new JsonSerializer<>
(), new JsonDeserializer<>(EnrichedOrder.class));

        return streamsBuilder
            .stream("orders", Consumed.with(Serdes.String(), orderSerde))

            // ☑ GOOD - Proper error handling
            .mapValues((readOnlyKey, order) -> {
                try {
                    return enrichOrder(order);
                } catch (Exception e) {
                    log.error("Error enriching order: {}", order.getOrderId(), e);
                    // Return a default/error state instead of throwing
                    return createErrorOrder(order, e);
                }
            })

            // ☑ GOOD - Filter out error orders
            .filter((key, order) -> !order.isError())

            // ☑ GOOD - Proper branching with cleanup
```

```
            .split(Named.as("order-processing-"))
            .branch((key, order) -> order.getAmount().compareTo(new
BigDecimal("1000")) > 0,
                    Branched.<String, OrderEvent>as("high-value")
                        .withConsumer(stream -> stream.to("high-value-orders",
                            Produced.with(Serdes.String(), orderSerde))))
            .defaultBranch(Branched.<String, OrderEvent>as("standard")
                        .withConsumer(stream -> stream.to("standard-orders",
                            Produced.with(Serdes.String(), orderSerde))));
    }

    private OrderEvent enrichOrder(OrderEvent order) {
        // Safe enrichment logic
        return order.toBuilder()
            .processedTimestamp(Instant.now())
            .build();
    }

    private OrderEvent createErrorOrder(OrderEvent original, Exception error) {
        return original.toBuilder()
            .error(true)
            .errorMessage(error.getMessage())
            .build();
    }
}
```

☑ Multi-tenancy Excellence

```
/**
 * ☑ GOOD - Secure multi-tenant implementation
 */
@Service
@lombok.extern.slf4j.Slf4j
public class ProductionMultiTenantService {

    @Autowired
    private TenantSecurityService tenantSecurity;

    @Autowired
    private MultiTenantKafkaService kafkaService;

    /**
     * ☑ GOOD - Proper tenant validation and security
     */
    @PreAuthorize("@tenantSecurity.hasAccessToTenant(authentication, #tenantId)")
    public void sendSecureMessage(String tenantId, String topic, Object message) {

        // Validate tenant exists and is active
        if (!tenantSecurity.isTenantActive(tenantId)) {
            throw new IllegalArgumentException("Tenant is not active: " +
tenantId);
```

```
        }

        // Set secure tenant context
        try {
            TenantContextHolder.setTenantId(tenantId);

            // Add audit information
            Map<String, Object> auditedMessage = new HashMap<>();
            auditedMessage.put("originalMessage", message);
            auditedMessage.put("tenantId", tenantId);
            auditedMessage.put("timestamp", Instant.now());
            auditedMessage.put("userId",
SecurityContextHolder.getContext().getAuthentication().getName());

            kafkaService.sendTenantMessage(topic, UUID.randomUUID().toString(),
auditedMessage);

            log.info("Sent secure message: tenant={}, topic={}", tenantId, topic);

        } finally {
            // ☑ CRITICAL - Always clear context
            TenantContextHolder.clear();
        }
    }

    /**
     * ☑ GOOD - Tenant-specific configuration with caching
     */
    @Cacheable(value = "tenantConfigs", key = "#tenantId")
    public TenantConfiguration getTenantConfiguration(String tenantId) {

        // Validate access
        if (!tenantSecurity.hasAccessToTenant(
                SecurityContextHolder.getContext().getAuthentication(), tenantId))
{
            throw new SecurityException("No access to tenant configuration: " +
tenantId);
        }

        return loadTenantConfigurationSecurely(tenantId);
    }

    private TenantConfiguration loadTenantConfigurationSecurely(String tenantId) {
        // Load from secure configuration store
        return TenantConfiguration.builder()
            .tenantId(tenantId)
            .tier(determineTenantTier(tenantId))
            .processingEnabled(true)
            .build();
    }

    private String determineTenantTier(String tenantId) {
        // Determine tenant tier based on subscription or configuration
        return "STANDARD";
```

```java
    }
}

/**
 * ☑ GOOD - Tenant security service
 */
@Service
public class TenantSecurityService {

    public boolean hasAccessToTenant(Authentication auth, String tenantId) {
        // Implement proper tenant access validation
        // Check user permissions, tenant membership, etc.
        return auth != null && auth.isAuthenticated();
    }

    public boolean isTenantActive(String tenantId) {
        // Check if tenant is active and in good standing
        return true; // Implement proper tenant status check
    }
}
```

📈 Version Highlights

Spring Kafka Advanced Features Timeline

Version	Release Date	Key Advanced Features
Spring Kafka 3.3.x	2024	**Enhanced ReplyingKafkaTemplate**, improved Streams integration
Spring Kafka 3.2.x	2024	**Multi-tenancy patterns**, better Connect integration
Spring Kafka 3.1.x	2023	**Kafka Streams auto-config**, enhanced request/reply patterns
Spring Kafka 3.0.x	2022	**Observation API**, modern streams support
Spring Kafka 2.9.x	2022	**Interactive Queries**, improved Streams DSL
Spring Kafka 2.8.x	2022	**Enhanced ReplyingKafkaTemplate**, better error handling
Spring Kafka 2.7.x	2021	**Message<?> support** in ReplyingKafkaTemplate
Spring Kafka 2.1.3	2018	**ReplyingKafkaTemplate introduction**

Feature Maturity Matrix

```
Feature Maturity Timeline:

ReplyingKafkaTemplate:
2018 ═══════════════════════════════════ Stable
     Initial Release → Message Support → Production Ready

Kafka Streams Integration:
```

```
2021 ██████████████████████████ Mature
     Basic Support → Auto-Config → Interactive Queries → Production

Kafka Connect Integration:
2022 ═══════════════════ Developing
     Manual Config → REST Client → Management APIs → Automation

Multi-tenancy Support:
2023 ═══════ Emerging
     Basic Patterns → Security → Advanced Routing → Production
```

Production Readiness Guide

Deployment Checklist

Request/Reply Systems:

- ☑ **Configure appropriate timeouts** (30 seconds max for most use cases)
- ☑ **Implement circuit breakers** for downstream service protection
- ☑ **Set up monitoring** for request/reply latency and success rates
- ☑ **Plan for reply topic scaling** (partitions = max concurrent requests)
- ☑ **Implement correlation ID tracking** for debugging

Kafka Streams Applications:

- ☑ **Size state stores properly** (estimate state size, configure RocksDB)
- ☑ **Plan for rebalancing** (minimize during business hours)
- ☑ **Set up monitoring** for lag, throughput, and error rates
- ☑ **Configure changelog topics** for fault tolerance
- ☑ **Implement proper error handling** in stream processors

Multi-tenant Systems:

- ☑ **Implement tenant isolation** at topic and security levels
- ☑ **Set up tenant-specific monitoring** and alerting
- ☑ **Plan for tenant onboarding/offboarding** automation
- ☑ **Implement proper access controls** and audit logging
- ☑ **Design for tenant scalability** (resource allocation per tenant)

Performance Optimization:

- **ReplyingKafkaTemplate**: Use connection pooling, optimize serialization
- **Kafka Streams**: Tune commit intervals, cache sizes, thread counts
- **Multi-tenancy**: Implement tenant-aware caching, optimize topic routing
- **General**: Monitor JVM heap, GC performance, network usage

This comprehensive Spring Kafka Advanced Features guide provides production-ready patterns for implementing sophisticated Kafka applications, from synchronous request/reply systems to real-time stream processing, data integration pipelines, and multi-tenant architectures, ensuring scalability and operational excellence in complex enterprise environments.

[685] [686] [687]