Spring Kafka Security: Part 2 - Spring Boot Configuration & Production Guide

Continuation of the comprehensive Spring Kafka Security guide covering Spring Boot property-based configuration, authorization, best practices, and production security patterns.

⚙️ Spring Boot Property-Based Security Configuration

Simple Explanation: Spring Boot provides declarative security configuration through `application.yml` or `application.properties` files, eliminating the need for programmatic bean configuration. This approach is cleaner, more maintainable, and supports externalized configuration for different environments.

Spring Boot Security Property Structure:

```
# Complete Spring Boot Kafka Security Configuration Structure

spring:
  kafka:
    # Basic connection settings
    bootstrap-servers: localhost:9093
    client-id: secure-spring-app

    # SSL Configuration
    ssl:
      key-password: key-secret
      key-store-location: classpath:ssl/kafka.client.keystore.jks
      key-store-password: keystore-secret
      key-store-type: JKS
      trust-store-location: classpath:ssl/kafka.client.truststore.jks
      trust-store-password: truststore-secret
      trust-store-type: JKS
      protocol: TLSv1.3

    # Security Protocol
    security:
      protocol: SASL_SSL  # Options: PLAINTEXT, SSL, SASL_PLAINTEXT, SASL_SSL

    # SASL Configuration
    sasl:
      mechanism: SCRAM-SHA-256  # Options: PLAIN, SCRAM-SHA-256, SCRAM-SHA-512,
GSSAPI
      jaas:
        config: |
          org.apache.kafka.common.security.scram.ScramLoginModule required
          username="secure-user"
          password="secure-password";

    # Producer Configuration
```

```
    producer:
      acks: all
      retries: 2147483647
      key-serializer: org.apache.kafka.common.serialization.StringSerializer
      value-serializer:
org.springframework.kafka.support.serializer.JsonSerializer
      batch-size: 65536
      linger-ms: 10
      compression-type: snappy

    # Consumer Configuration
    consumer:
      group-id: secure-consumer-group
      auto-offset-reset: earliest
      enable-auto-commit: false
      key-deserializer: org.apache.kafka.common.serialization.StringDeserializer
      value-deserializer:
org.springframework.kafka.support.serializer.JsonDeserializer
      max-poll-records: 100
      session-timeout-ms: 30000
      heartbeat-interval-ms: 10000

    # Listener Configuration
    listener:
      ack-mode: manual_immediate
      concurrency: 3
      type: batch  # Options: single, batch
```

Environment-Specific Security Configuration

```
# application.yml - Base configuration
spring:
  kafka:
    bootstrap-servers: ${KAFKA_BOOTSTRAP_SERVERS:localhost:9092}
    security:
      protocol: ${KAFKA_SECURITY_PROTOCOL:SASL_SSL}
    sasl:
      mechanism: ${KAFKA_SASL_MECHANISM:SCRAM-SHA-256}

# application-dev.yml - Development environment
spring:
  kafka:
    bootstrap-servers: localhost:9092
    security:
      protocol: SASL_PLAINTEXT  # No SSL for local development
    sasl:
      mechanism: PLAIN
      jaas:
        config: |
          org.apache.kafka.common.security.plain.PlainLoginModule required
          username="dev-user"
```

```
          password="dev-password";

# application-staging.yml - Staging environment
spring:
  kafka:
    bootstrap-servers: kafka-staging.company.com:9093
    security:
      protocol: SASL_SSL
    ssl:
      trust-store-location: ${KAFKA_SSL_TRUSTSTORE_PATH:/etc/ssl/kafka-
truststore.jks}
      trust-store-password: ${KAFKA_SSL_TRUSTSTORE_PASSWORD}
      key-store-location: ${KAFKA_SSL_KEYSTORE_PATH:/etc/ssl/kafka-keystore.jks}
      key-store-password: ${KAFKA_SSL_KEYSTORE_PASSWORD}
      key-password: ${KAFKA_SSL_KEY_PASSWORD}
    sasl:
      mechanism: SCRAM-SHA-256
      jaas:
        config: |
          org.apache.kafka.common.security.scram.ScramLoginModule required
          username="${KAFKA_SASL_USERNAME}"
          password="${KAFKA_SASL_PASSWORD}";

# application-prod.yml - Production environment
spring:
  kafka:
    bootstrap-servers: kafka-prod-1.company.com:9093,kafka-prod-
2.company.com:9093,kafka-prod-3.company.com:9093
    security:
      protocol: SASL_SSL
    ssl:
      trust-store-location: file:${KAFKA_SSL_TRUSTSTORE_PATH}
      trust-store-password: ${KAFKA_SSL_TRUSTSTORE_PASSWORD}
      key-store-location: file:${KAFKA_SSL_KEYSTORE_PATH}
      key-store-password: ${KAFKA_SSL_KEYSTORE_PASSWORD}
      key-password: ${KAFKA_SSL_KEY_PASSWORD}
      protocol: TLSv1.3
      enabled-protocols: TLSv1.3,TLSv1.2
      endpoint-identification-algorithm: https
    sasl:
      mechanism: SCRAM-SHA-512  # Stronger hashing for production
      jaas:
        config: |
          org.apache.kafka.common.security.scram.ScramLoginModule required
          username="${KAFKA_SASL_USERNAME}"
          password="${KAFKA_SASL_PASSWORD}";
    producer:
      acks: all
      retries: 2147483647
      max-in-flight-requests-per-connection: 1  # For strict ordering
      enable-idempotence: true
      batch-size: 65536
      linger-ms: 10
      compression-type: snappy
```

```
      request-timeout-ms: 30000
      delivery-timeout-ms: 120000
    consumer:
      auto-offset-reset: earliest
      enable-auto-commit: false
      isolation-level: read_committed
      max-poll-records: 50  # Smaller batches for production
      session-timeout-ms: 30000
      heartbeat-interval-ms: 10000
      fetch-min-bytes: 50000
      fetch-max-wait-ms: 500
```

Spring Boot SSL Bundle Configuration (Spring Boot 3.1+)

```
# Modern SSL Bundle approach (Spring Boot 3.1+)
spring:
  ssl:
    bundle:
      jks:
        kafka-client:
          key:
            alias: client
            password: ${KAFKA_SSL_KEY_PASSWORD}
          keystore:
            location: ${KAFKA_SSL_KEYSTORE_PATH}
            password: ${KAFKA_SSL_KEYSTORE_PASSWORD}
            type: JKS
          truststore:
            location: ${KAFKA_SSL_TRUSTSTORE_PATH}
            password: ${KAFKA_SSL_TRUSTSTORE_PASSWORD}
            type: JKS

        kafka-server:
          keystore:
            location: ${KAFKA_SSL_SERVER_KEYSTORE_PATH}
            password: ${KAFKA_SSL_SERVER_KEYSTORE_PASSWORD}
          truststore:
            location: ${KAFKA_SSL_SERVER_TRUSTSTORE_PATH}
            password: ${KAFKA_SSL_SERVER_TRUSTSTORE_PASSWORD}

      pem:
        kafka-pem-client:
          keystore:
            certificate: ${KAFKA_SSL_CERT_PATH}
            private-key: ${KAFKA_SSL_KEY_PATH}
          truststore:
            certificate: ${KAFKA_SSL_CA_CERT_PATH}

  kafka:
    ssl:
      bundle: kafka-client  # Reference to SSL bundle
```

Comprehensive Spring Boot Configuration Class

```
import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.boot.context.properties.EnableConfigurationProperties;
import org.springframework.context.annotation.Configuration;

/**
 * Spring Boot property-based security configuration
 */
@Configuration
@EnableConfigurationProperties({
    KafkaSecurityProperties.class,
    KafkaSSLProperties.class,
    KafkaSASLProperties.class
})
@lombok.extern.slf4j.Slf4j
public class SpringBootKafkaSecurityConfiguration {

    // Configuration is handled automatically by Spring Boot
    // No manual bean definitions required when using properties

    @PostConstruct
    public void logSecurityConfiguration() {
        log.info("Spring Boot Kafka security configuration enabled");
        log.info("Security protocol: {}", kafkaSecurityProperties.getProtocol());
        log.info("SASL mechanism: {}", kafkaSaslProperties.getMechanism());
        log.info("SSL enabled: {}", kafkaSSLProperties.isEnabled());
    }

    @Autowired
    private KafkaSecurityProperties kafkaSecurityProperties;

    @Autowired
    private KafkaSSLProperties kafkaSSLProperties;

    @Autowired
    private KafkaSASLProperties kafkaSaslProperties;
}

/**
 * Custom security properties for additional configuration
 */
@ConfigurationProperties(prefix = "kafka.security")
@lombok.Data
public class KafkaSecurityProperties {

    private String protocol = "SASL_SSL";
    private boolean enforceSSL = true;
    private boolean enableHostnameVerification = true;
    private int connectionTimeoutMs = 30000;
    private int requestTimeoutMs = 30000;
```

```
    /**
     * ACL configuration
     */
    private ACLProperties acl = new ACLProperties();

    @lombok.Data
    public static class ACLProperties {
        private boolean enabled = false;
        private String defaultDenyPolicy = "DENY";
        private List<String> superUsers = new ArrayList<>();
        private boolean allowEveryoneIfNoAclFound = false;
    }
}

/**
 * Custom SSL properties
 */
@ConfigurationProperties(prefix = "kafka.ssl")
@lombok.Data
public class KafkaSSLProperties {

    private boolean enabled = true;
    private String protocol = "TLSv1.3";
    private List<String> enabledProtocols = Arrays.asList("TLSv1.3", "TLSv1.2");
    private String endpointIdentificationAlgorithm = "https";
    private String provider = "";
    private List<String> cipherSuites = new ArrayList<>();

    /**
     * Certificate validation settings
     */
    private ValidationProperties validation = new ValidationProperties();

    @lombok.Data
    public static class ValidationProperties {
        private boolean validateCertificateChain = true;
        private boolean validateHostname = true;
        private boolean allowSelfSignedCertificates = false;
        private int certificateExpirationWarningDays = 30;
    }
}

/**
 * Custom SASL properties
 */
@ConfigurationProperties(prefix = "kafka.sasl")
@lombok.Data
public class KafkaSASLProperties {

    private String mechanism = "SCRAM-SHA-256";
    private String username;
    private String password;
    private int loginRefreshWindowFactor = 80;
```

```
    private int loginRefreshMinPeriodSeconds = 60;
    private boolean loginRefreshBufferSeconds = 300;

    /**
     * SCRAM-specific properties
     */
    private SCRAMProperties scram = new SCRAMProperties();

    @lombok.Data
    public static class SCRAMProperties {
        private int iterations = 8192;
        private String hashAlgorithm = "SHA-256";
        private boolean enableTokenRefresh = true;
        private int tokenLifetimeMs = 3600000; // 1 hour
    }
}

/**
 * Property-based message producer using Spring Boot configuration
 */
@Service
@lombok.extern.slf4j.Slf4j
public class PropertyBasedMessageProducer {

    @Autowired
    private KafkaTemplate<String, Object> kafkaTemplate; // Auto-configured by
Spring Boot

    @Value("${spring.kafka.security.protocol}")
    private String securityProtocol;

    /**
     * Send secure message using Spring Boot auto-configuration
     */
    public void sendSecureMessageWithProperties(String topic, String key, Object
message) {

        log.info("Sending secure message with Spring Boot properties: topic={},
key={}, security={}",
            topic, key, securityProtocol);

        try {
            // KafkaTemplate is auto-configured with security properties
            ListenableFuture<SendResult<String, Object>> future =
                kafkaTemplate.send(topic, key, message);

            future.addCallback(
                result -> log.info("Property-based secure message sent: offset=
{}",
                    result.getRecordMetadata().offset()),
                failure -> log.error("Failed to send property-based secure
message", failure)
            );
```

```
            } catch (Exception e) {
                log.error("Error sending property-based secure message", e);
                throw e;
            }
        }

        /**
         * Send batch of secure messages
         */
        public void sendSecureMessageBatch(String topic, Map<String, Object> messages)
{

            log.info("Sending secure message batch with properties: topic={}, count=
{}",
                topic, messages.size());

            List<ListenableFuture<SendResult<String, Object>>> futures = new
ArrayList<>();

            for (Map.Entry<String, Object> entry : messages.entrySet()) {
                ListenableFuture<SendResult<String, Object>> future =
                    kafkaTemplate.send(topic, entry.getKey(), entry.getValue());
                futures.add(future);
            }

            // Wait for all to complete
            futures.forEach(future -> {
                try {
                    SendResult<String, Object> result = future.get(30,
TimeUnit.SECONDS);
                    log.debug("Batch message sent: offset={}",
result.getRecordMetadata().offset());
                } catch (Exception e) {
                    log.error("Failed to send batch message", e);
                }
            });

            log.info("Secure message batch completed: topic={}, count={}", topic,
messages.size());
        }
}

/**
 * Property-based message consumer using Spring Boot configuration
 */
@Component
@lombok.extern.slf4j.Slf4j
public class PropertyBasedMessageConsumer {

    /**
     * Consume secure messages using Spring Boot auto-configuration
     */
    @KafkaListener(
        topics = "${app.kafka.secure-topic:secure-topic}",
```

```
        groupId = "${spring.kafka.consumer.group-id}",
        concurrency = "${spring.kafka.listener.concurrency:3}"
    )
    public void consumeSecureMessageWithProperties(@Payload String message,

@Header(KafkaHeaders.RECEIVED_TOPIC) String topic,

@Header(KafkaHeaders.RECEIVED_PARTITION) int partition,
                                                   @Header(KafkaHeaders.OFFSET) long
offset,
                                                   Acknowledgment ack) {

        log.info("Received secure message with Spring Boot properties: topic={},
partition={}, offset={}",
            topic, partition, offset);

        try {
            // Process secure message
            processSecureMessage(message);

            // Manual acknowledgment (configured via properties)
            if (ack != null) {
                ack.acknowledge();
            }

            log.debug("Property-based secure message processed: offset={}",
offset);

        } catch (Exception e) {
            log.error("Error processing property-based secure message: topic={},
offset={}",
                topic, offset, e);
            throw e;
        }
    }

    /**
     * Batch consumer using property configuration
     */
    @KafkaListener(
        topics = "${app.kafka.batch-topic:batch-topic}",
        groupId = "${spring.kafka.consumer.group-id}-batch",
        containerFactory = "kafkaListenerContainerFactory"  // Auto-configured
    )
    public void consumeSecureMessageBatch(@Payload List<String> messages,
                                          @Header(KafkaHeaders.RECEIVED_TOPIC)
List<String> topics,
                                          @Header(KafkaHeaders.OFFSET) List<Long>
offsets,
                                          Acknowledgment ack) {

        log.info("Received secure message batch with properties: size={}, topics=
{}",
            messages.size(),
```

```
topics.stream().distinct().collect(Collectors.toList()));

        try {
            // Process batch
            for (int i = 0; i < messages.size(); i++) {
                String message = messages.get(i);
                String topic = topics.get(i);
                Long offset = offsets.get(i);

                log.debug("Processing batch message: topic={}, offset={}", topic,
offset);
                processSecureMessage(message);
            }

            // Batch acknowledgment
            if (ack != null) {
                ack.acknowledge();
            }

            log.info("Property-based secure message batch processed: size={}",
messages.size());

        } catch (Exception e) {
            log.error("Error processing property-based secure message batch", e);
            throw e;
        }
    }

    private void processSecureMessage(String message) {
        log.debug("Processing secure message: {}", message);
        // Business logic here
    }
}

/**
 * Configuration validation service
 */
@Component
@lombok.extern.slf4j.Slf4j
public class KafkaSecurityConfigurationValidator {

    @Autowired
    private KafkaProperties kafkaProperties;

    @Autowired(required = false)
    private KafkaSecurityProperties securityProperties;

    @EventListener(ApplicationReadyEvent.class)
    public void validateSecurityConfiguration() {

        log.info("Validating Kafka security configuration...");

        List<String> warnings = new ArrayList<>();
        List<String> errors = new ArrayList<>();
```

```
        // Validate security protocol
        String securityProtocol = kafkaProperties.getSecurity().getProtocol();
        if ("PLAINTEXT".equals(securityProtocol)) {
            warnings.add("Using PLAINTEXT protocol - data is not encrypted");
        }
        if ("SASL_PLAINTEXT".equals(securityProtocol)) {
            warnings.add("Using SASL_PLAINTEXT - SASL credentials are not
encrypted");
        }

        // Validate SSL configuration
        if ("SSL".equals(securityProtocol) || "SASL_SSL".equals(securityProtocol))
{
            KafkaProperties.Ssl ssl = kafkaProperties.getSsl();

            if (ssl.getTrustStoreLocation() == null) {
                errors.add("SSL truststore location is required for
SSL/SASL_SSL");
            }

            if (ssl.getKeyStoreLocation() == null &&
                (ssl.getKeyPassword() != null || ssl.getKeyStorePassword() !=
null)) {
                warnings.add("SSL keystore not configured but key passwords
provided");
            }

            if ("".equals(ssl.getEndpointIdentificationAlgorithm())) {
                warnings.add("SSL hostname verification is disabled");
            }
        }

        // Validate SASL configuration
        if ("SASL_PLAINTEXT".equals(securityProtocol) ||
"SASL_SSL".equals(securityProtocol)) {
            KafkaProperties.Sasl sasl = kafkaProperties.getSasl();

            if (sasl.getJaas() == null || sasl.getJaas().getConfig() == null) {
                errors.add("SASL JAAS configuration is required for SASL
protocols");
            }

            if ("PLAIN".equals(sasl.getMechanism()) &&
!"SASL_SSL".equals(securityProtocol)) {
                warnings.add("PLAIN mechanism without SSL encryption exposes
credentials");
            }
        }

        // Log results
        if (!errors.isEmpty()) {
            log.error("Kafka security configuration errors:");
            errors.forEach(error -> log.error("  - {}", error));
```

```
            throw new IllegalStateException("Kafka security configuration is
invalid");
        }

        if (!warnings.isEmpty()) {
            log.warn("Kafka security configuration warnings:");
            warnings.forEach(warning -> log.warn("  - {}", warning));
        }

        if (errors.isEmpty() && warnings.isEmpty()) {
            log.info("Kafka security configuration validation passed");
        }

        log.info("Security Protocol: {}", securityProtocol);
        if (kafkaProperties.getSasl() != null) {
            log.info("SASL Mechanism: {}",
kafkaProperties.getSasl().getMechanism());
        }
        log.info("SSL Enabled: {}",
            "SSL".equals(securityProtocol) ||
"SASL_SSL".equals(securityProtocol));
    }
}
```

Docker Compose with Environment Variables

```
# docker-compose.yml - Production Kafka with security
version: '3.8'

services:
  zookeeper:
    image: confluentinc/cp-zookeeper:7.5.0
    hostname: zookeeper
    container_name: zookeeper
    ports:
      - "2181:2181"
    environment:
      ZOOKEEPER_CLIENT_PORT: 2181
      ZOOKEEPER_TICK_TIME: 2000
      ZOOKEEPER_SECURE_CLIENT_PORT: 2182
      ZOOKEEPER_AUTH_PROVIDER: sasl
      KAFKA_OPTS: "-
Djava.security.auth.login.config=/etc/kafka/secrets/zookeeper_jaas.conf"
    volumes:
      - ./secrets:/etc/kafka/secrets
      - ./ssl:/etc/ssl/certs

  kafka:
    image: confluentinc/cp-kafka:7.5.0
    hostname: kafka
    container_name: kafka
```

```
    depends_on:
      - zookeeper
    ports:
      - "9092:9092"
      - "9093:9093"
    environment:
      KAFKA_BROKER_ID: 1
      KAFKA_ZOOKEEPER_CONNECT: 'zookeeper:2181'
      KAFKA_LISTENER_SECURITY_PROTOCOL_MAP: PLAINTEXT:PLAINTEXT,SASL_SSL:SASL_SSL
      KAFKA_ADVERTISED_LISTENERS:
PLAINTEXT://localhost:9092,SASL_SSL://localhost:9093
      KAFKA_INTER_BROKER_LISTENER_NAME: SASL_SSL

      # SASL Configuration
      KAFKA_SASL_MECHANISM_INTER_BROKER_PROTOCOL: SCRAM-SHA-256
      KAFKA_SASL_ENABLED_MECHANISMS: PLAIN,SCRAM-SHA-256,SCRAM-SHA-512

      # SSL Configuration
      KAFKA_SSL_KEYSTORE_FILENAME: kafka.server.keystore.jks
      KAFKA_SSL_KEYSTORE_CREDENTIALS: keystore_credentials
      KAFKA_SSL_KEY_CREDENTIALS: key_credentials
      KAFKA_SSL_TRUSTSTORE_FILENAME: kafka.server.truststore.jks
      KAFKA_SSL_TRUSTSTORE_CREDENTIALS: truststore_credentials
      KAFKA_SSL_ENDPOINT_IDENTIFICATION_ALGORITHM: " "

      # Security
      KAFKA_SECURITY_INTER_BROKER_PROTOCOL: SASL_SSL
      KAFKA_SUPER_USERS: User:admin;User:kafka
      KAFKA_ALLOW_EVERYONE_IF_NO_ACL_FOUND: "false"
      KAFKA_AUTHORIZER_CLASS_NAME: kafka.security.authorizer.AclAuthorizer

      # JAAS Configuration
      KAFKA_OPTS: "-
Djava.security.auth.login.config=/etc/kafka/secrets/kafka_server_jaas.conf"

    volumes:
      - ./secrets:/etc/kafka/secrets
      - ./ssl:/etc/ssl/certs

  spring-app:
    build: .
    container_name: spring-kafka-app
    depends_on:
      - kafka
    environment:
      # Spring Boot Kafka Configuration
      SPRING_KAFKA_BOOTSTRAP_SERVERS: kafka:9093
      SPRING_KAFKA_SECURITY_PROTOCOL: SASL_SSL
      SPRING_KAFKA_SASL_MECHANISM: SCRAM-SHA-256
      SPRING_KAFKA_SASL_JAAS_CONFIG: |
        org.apache.kafka.common.security.scram.ScramLoginModule required
        username="spring-app-user"
        password="spring-app-password";
```

```
      # SSL Configuration
      SPRING_KAFKA_SSL_TRUST_STORE_LOCATION:
/etc/ssl/certs/kafka.client.truststore.jks
      SPRING_KAFKA_SSL_TRUST_STORE_PASSWORD: truststore-password
      SPRING_KAFKA_SSL_KEY_STORE_LOCATION:
/etc/ssl/certs/kafka.client.keystore.jks
      SPRING_KAFKA_SSL_KEY_STORE_PASSWORD: keystore-password
      SPRING_KAFKA_SSL_KEY_PASSWORD: key-password

      # Consumer/Producer Configuration
      SPRING_KAFKA_CONSUMER_GROUP_ID: secure-spring-app
      SPRING_KAFKA_CONSUMER_AUTO_OFFSET_RESET: earliest
      SPRING_KAFKA_CONSUMER_ENABLE_AUTO_COMMIT: false
      SPRING_KAFKA_PRODUCER_ACKS: all
      SPRING_KAFKA_PRODUCER_RETRIES: 2147483647

    volumes:
      - ./ssl:/etc/ssl/certs
    ports:
      - "8080:8080"
```

JAAS Configuration Files

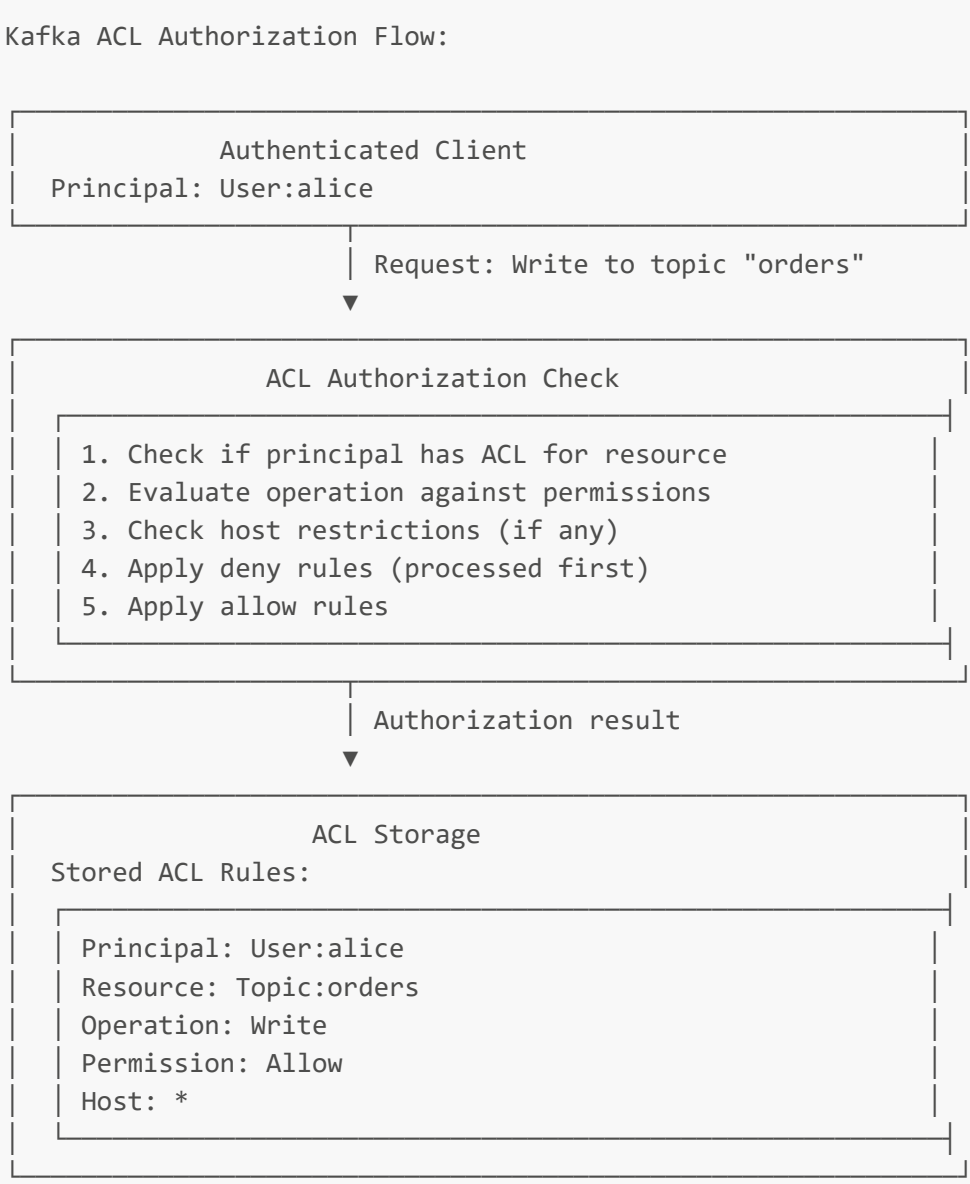
```
# kafka_server_jaas.conf
KafkaServer {
    org.apache.kafka.common.security.scram.ScramLoginModule required
    username="kafka"
    password="kafka-secret";
};

Client {
    org.apache.kafka.common.security.scram.ScramLoginModule required
    username="kafka"
    password="kafka-secret";
};

# zookeeper_jaas.conf
Server {
    org.apache.kafka.common.security.plain.PlainLoginModule required
    username="admin"
    password="admin-secret"
    user_admin="admin-secret"
    user_kafka="kafka-secret";
};
```

🔐 Authorization with Access Control Lists (ACLs)

Simple Explanation: ACLs (Access Control Lists) provide fine-grained authorization in Kafka, controlling which authenticated users can perform specific operations (read, write, create, delete) on specific resources (topics,

consumer groups, clusters).

ACL Architecture:



ACL Components:

Principal	Resource	Operation	Permission	Host
User:alice	Topic:orders	Write	Allow	*
User:bob	Topic:orders	Read	Allow	10.0.0.0/24
User:eve	Topic:*	*	Deny	*

ACL Management Service

```
/**
 * Comprehensive ACL management service
 */
```

```
@Service
@lombok.extern.slf4j.Slf4j
public class KafkaACLManagementService {

    @Autowired
    private AdminClient adminClient;

    /**
     * Create producer ACLs for a user
     */
    public void createProducerACLs(String username, String topicName) {

        log.info("Creating producer ACLs: user={}, topic={}", username,
topicName);

        try {
            KafkaPrincipal principal = new
KafkaPrincipal(KafkaPrincipal.USER_TYPE, username);

            List<AclBinding> aclBindings = Arrays.asList(
                // Write permission for topic
                new AclBinding(
                    new ResourcePattern(ResourceType.TOPIC, topicName,
PatternType.LITERAL),
                    new AccessControlEntry(principal.toString(), "*",
AclOperation.WRITE, AclPermissionType.ALLOW)
                ),

                // Describe permission for topic (needed for metadata)
                new AclBinding(
                    new ResourcePattern(ResourceType.TOPIC, topicName,
PatternType.LITERAL),
                    new AccessControlEntry(principal.toString(), "*",
AclOperation.DESCRIBE, AclPermissionType.ALLOW)
                ),

                // Create permission for topic (in case it doesn't exist)
                new AclBinding(
                    new ResourcePattern(ResourceType.TOPIC, topicName,
PatternType.LITERAL),
                    new AccessControlEntry(principal.toString(), "*",
AclOperation.CREATE, AclPermissionType.ALLOW)
                )
            );

            CreateAclsResult result = adminClient.createAcls(aclBindings);
            result.all().get(30, TimeUnit.SECONDS);

            log.info("Producer ACLs created successfully: user={}, topic={}",
username, topicName);

        } catch (Exception e) {
            log.error("Failed to create producer ACLs: user={}, topic={}",
username, topicName, e);
```

```
            throw new RuntimeException("Producer ACL creation failed", e);
        }
    }

    /**
     * Create consumer ACLs for a user
     */
    public void createConsumerACLs(String username, String topicName, String
consumerGroup) {

        log.info("Creating consumer ACLs: user={}, topic={}, group={}", username,
topicName, consumerGroup);

        try {
            KafkaPrincipal principal = new
KafkaPrincipal(KafkaPrincipal.USER_TYPE, username);

            List<AclBinding> aclBindings = Arrays.asList(
                // Read permission for topic
                new AclBinding(
                    new ResourcePattern(ResourceType.TOPIC, topicName,
PatternType.LITERAL),
                    new AccessControlEntry(principal.toString(), "*",
AclOperation.READ, AclPermissionType.ALLOW)
                ),

                // Describe permission for topic
                new AclBinding(
                    new ResourcePattern(ResourceType.TOPIC, topicName,
PatternType.LITERAL),
                    new AccessControlEntry(principal.toString(), "*",
AclOperation.DESCRIBE, AclPermissionType.ALLOW)
                ),

                // Read permission for consumer group
                new AclBinding(
                    new ResourcePattern(ResourceType.GROUP, consumerGroup,
PatternType.LITERAL),
                    new AccessControlEntry(principal.toString(), "*",
AclOperation.READ, AclPermissionType.ALLOW)
                )
            );

            CreateAclsResult result = adminClient.createAcls(aclBindings);
            result.all().get(30, TimeUnit.SECONDS);

            log.info("Consumer ACLs created successfully: user={}, topic={},
group={}",
                username, topicName, consumerGroup);

        } catch (Exception e) {
            log.error("Failed to create consumer ACLs: user={}, topic={}, group=
{}",
                username, topicName, consumerGroup, e);
```

```
            throw new RuntimeException("Consumer ACL creation failed", e);
        }
    }

    /**
     * Create admin ACLs for cluster management
     */
    public void createAdminACLs(String username) {

        log.info("Creating admin ACLs: user={}", username);

        try {
            KafkaPrincipal principal = new
KafkaPrincipal(KafkaPrincipal.USER_TYPE, username);

            List<AclBinding> aclBindings = Arrays.asList(
                // Cluster admin permissions
                new AclBinding(
                    new ResourcePattern(ResourceType.CLUSTER, "kafka-cluster",
PatternType.LITERAL),
                    new AccessControlEntry(principal.toString(), "*",
AclOperation.ALL, AclPermissionType.ALLOW)
                ),

                // All topics permissions
                new AclBinding(
                    new ResourcePattern(ResourceType.TOPIC, "*",
PatternType.LITERAL),
                    new AccessControlEntry(principal.toString(), "*",
AclOperation.ALL, AclPermissionType.ALLOW)
                ),

                // All consumer groups permissions
                new AclBinding(
                    new ResourcePattern(ResourceType.GROUP, "*",
PatternType.LITERAL),
                    new AccessControlEntry(principal.toString(), "*",
AclOperation.ALL, AclPermissionType.ALLOW)
                )
            );

            CreateAclsResult result = adminClient.createAcls(aclBindings);
            result.all().get(30, TimeUnit.SECONDS);

            log.info("Admin ACLs created successfully: user={}", username);

        } catch (Exception e) {
            log.error("Failed to create admin ACLs: user={}", username, e);
            throw new RuntimeException("Admin ACL creation failed", e);
        }
    }

    /**
     * Create prefix-based ACLs for multiple topics
```

```
     */
    public void createPrefixACLs(String username, String topicPrefix, AclOperation
operation) {

        log.info("Creating prefix ACLs: user={}, prefix={}, operation={}",
username, topicPrefix, operation);

        try {
            KafkaPrincipal principal = new
KafkaPrincipal(KafkaPrincipal.USER_TYPE, username);

            AclBinding aclBinding = new AclBinding(
                new ResourcePattern(ResourceType.TOPIC, topicPrefix,
PatternType.PREFIXED),
                new AccessControlEntry(principal.toString(), "*", operation,
AclPermissionType.ALLOW)
            );

            CreateAclsResult result =
adminClient.createAcls(Collections.singletonList(aclBinding));
            result.all().get(30, TimeUnit.SECONDS);

            log.info("Prefix ACLs created successfully: user={}, prefix={},
operation={}",
                username, topicPrefix, operation);

        } catch (Exception e) {
            log.error("Failed to create prefix ACLs: user={}, prefix={},
operation={}",
                username, topicPrefix, operation, e);
            throw new RuntimeException("Prefix ACL creation failed", e);
        }
    }

    /**
     * List ACLs for a specific user
     */
    public List<AclBinding> listUserACLs(String username) {

        log.info("Listing ACLs for user: {}", username);

        try {
            KafkaPrincipal principal = new
KafkaPrincipal(KafkaPrincipal.USER_TYPE, username);

            AclBindingFilter filter = new AclBindingFilter(
                ResourcePatternFilter.ANY,
                new AccessControlEntryFilter(principal.toString(), null,
AclOperation.ANY, AclPermissionType.ANY)
            );

            DescribeAclsResult result = adminClient.describeAcls(filter);
            Collection<AclBinding> aclBindings = result.values().get(30,
TimeUnit.SECONDS);
```

```
            List<AclBinding> userAcls = new ArrayList<>(aclBindings);

            log.info("Found {} ACLs for user: {}", userAcls.size(), username);

            return userAcls;

        } catch (Exception e) {
            log.error("Failed to list ACLs for user: {}", username, e);
            throw new RuntimeException("ACL listing failed", e);
        }
    }

    /**
     * Delete ACLs for a user
     */
    public void deleteUserACLs(String username) {

        log.info("Deleting ACLs for user: {}", username);

        try {
            List<AclBinding> userAcls = listUserACLs(username);

            if (userAcls.isEmpty()) {
                log.info("No ACLs found for user: {}", username);
                return;
            }

            List<AclBindingFilter> filtersToDelete = userAcls.stream()
                .map(acl -> acl.toFilter())
                .collect(Collectors.toList());

            DeleteAclsResult result = adminClient.deleteAcls(filtersToDelete);
            result.all().get(30, TimeUnit.SECONDS);

            log.info("Deleted {} ACLs for user: {}", userAcls.size(), username);

        } catch (Exception e) {
            log.error("Failed to delete ACLs for user: {}", username, e);
            throw new RuntimeException("ACL deletion failed", e);
        }
    }

    /**
     * Check if user has permission for specific operation
     */
    public boolean hasPermission(String username, ResourceType resourceType,
String resourceName,
                                 AclOperation operation) {

        log.debug("Checking permission: user={}, resource={}:{}, operation={}",
            username, resourceType, resourceName, operation);

        try {
```

```
            List<AclBinding> userAcls = listUserACLs(username);

            for (AclBinding acl : userAcls) {
                ResourcePattern resource = acl.pattern();
                AccessControlEntry entry = acl.entry();

                // Check if ACL matches the requested resource and operation
                if (resource.resourceType() == resourceType &&
                    (resource.name().equals(resourceName) ||
resource.name().equals("*")) &&
                    (entry.operation() == operation || entry.operation() ==
AclOperation.ALL) &&
                    entry.permissionType() == AclPermissionType.ALLOW) {

                    log.debug("Permission granted: user={}, resource={}:{},
operation={}",
                        username, resourceType, resourceName, operation);
                    return true;
                }
            }

            log.debug("Permission denied: user={}, resource={}:{}, operation={}",
                username, resourceType, resourceName, operation);
            return false;

        } catch (Exception e) {
            log.error("Failed to check permission: user={}, resource={}:{},
operation={}",
                username, resourceType, resourceName, operation, e);
            return false;
        }
    }
}
```

This completes Part 2 of the Spring Kafka Security guide, covering Spring Boot property-based configuration and ACL authorization. The guide continues with comparisons, best practices, and production patterns in the final part.