Spring Kafka Security: Complete Developer Guide

A comprehensive guide covering all aspects of Spring Kafka security, including SSL/TLS encryption, SASL authentication mechanisms, Spring Boot property-based configuration, and production security patterns with extensive Java examples.

Table of Contents

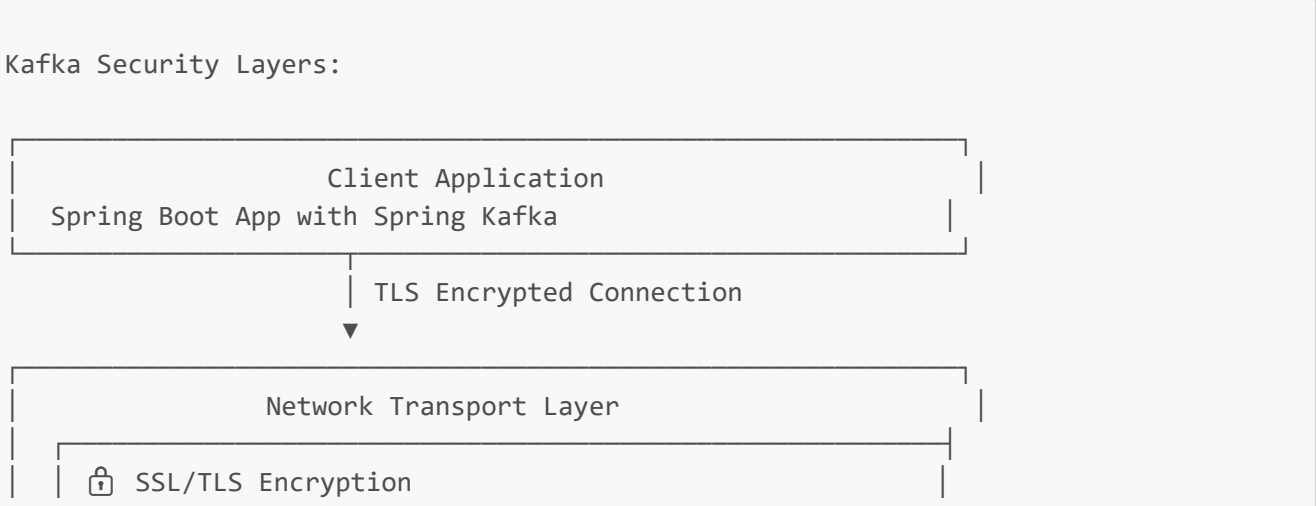
What is Kafka Security?

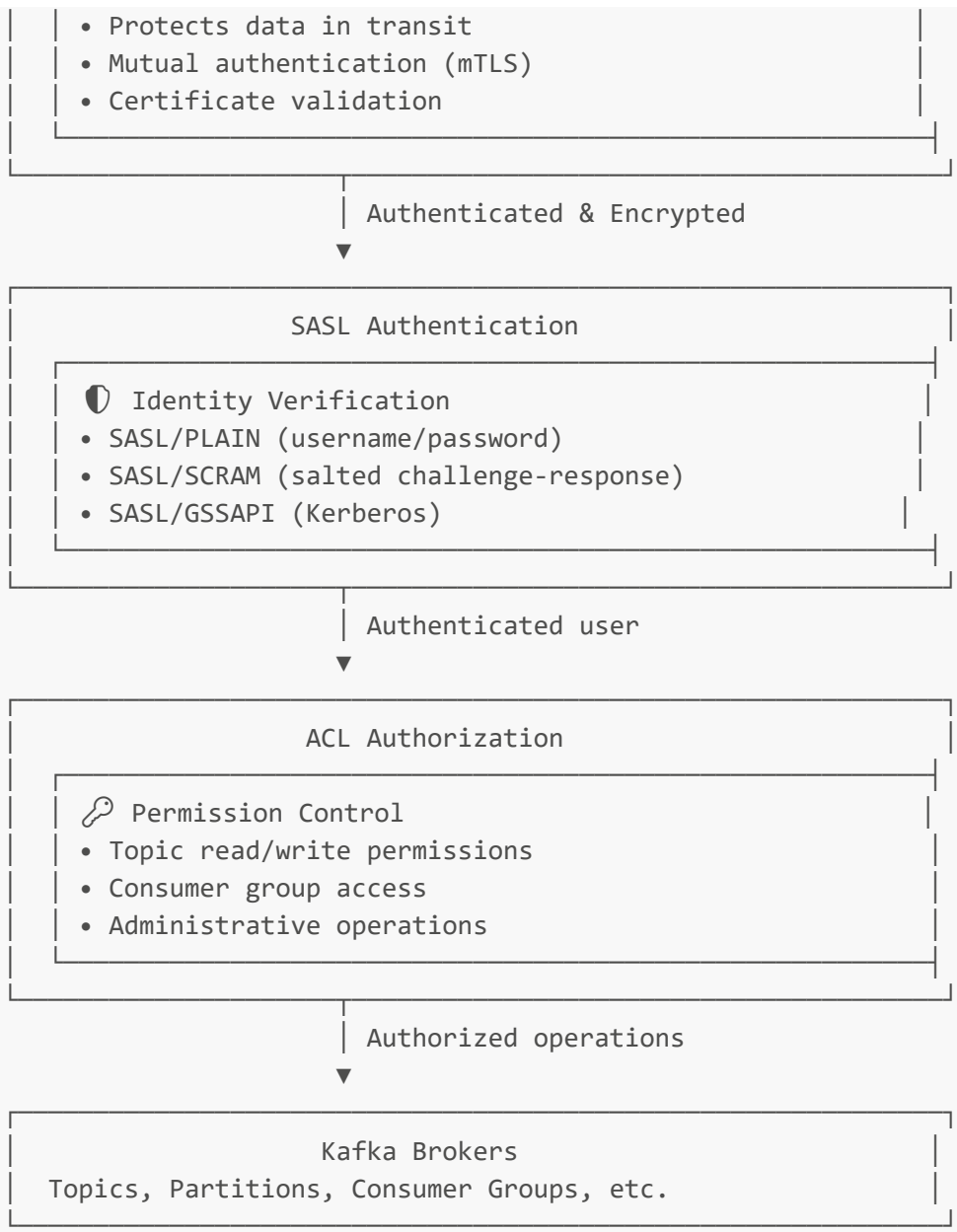
Simple Explanation: Kafka security protects your data streams through three key mechanisms: **Authentication** (who can connect), **Authorization** (what they can do), and **Encryption** (protecting data in transit). This ensures that only authorized clients can access your Kafka cluster and that sensitive data remains protected.

Why Security is Critical in Kafka:

- **Data Protection**: Prevents unauthorized access to sensitive message streams
- **Compliance Requirements**: Meets regulatory standards (GDPR, HIPAA, SOX)
- **Multi-Tenant Environments**: Isolates different applications and teams
- **Production Safety**: Prevents accidental or malicious data corruption
- **Audit Trail**: Tracks who accessed what data and when

Kafka Security Architecture:





Security Protocol Matrix:

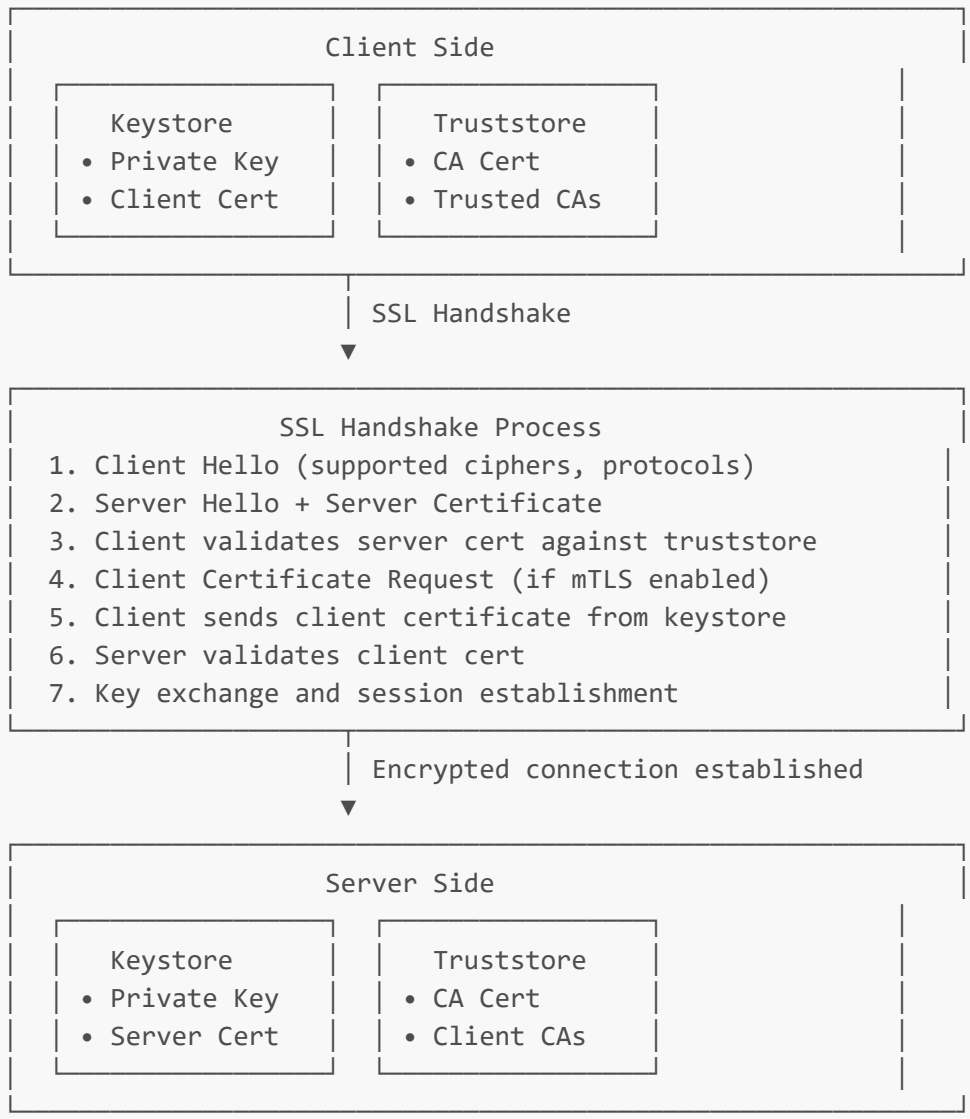
Protocol	Encryption	Auth Method	Use Case
PLAINTEXT	None	None	Development
SSL	TLS/SSL	mTLS	Simple Prod
SASL_PLAINTEXT	None	SASL	Testing
SASL_SSL	TLS/SSL	SASL + mTLS	Production

SSL/TLS Setup

Configuring truststore/keystore

Simple Explanation: SSL/TLS in Kafka uses certificates for encryption and authentication. The **truststore** contains certificates of trusted Certificate Authorities (CAs), while the **keystore** contains the client's private key and certificate for mutual authentication (mTLS).

SSL/TLS Certificate Flow:



Complete SSL/TLS Configuration with Spring Kafka

```
import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.kafka.core.*;
import org.springframework.kafka.config.ConcurrentKafkaListenerContainerFactory;

import org.apache.kafka.clients.CommonClientConfigs;
import org.apache.kafka.clients.consumer.ConsumerConfig;
import org.apache.kafka.clients.producer.ProducerConfig;
import org.apache.kafka.common.config.SslConfigs;
import org.apache.kafka.common.serialization.StringDeserializer;
import org.apache.kafka.common.serialization.StringSerializer;
```

```java
/**
 * Comprehensive SSL/TLS configuration for Spring Kafka
 */
@Configuration
@lombok.extern.slf4j.Slf4j
public class KafkaSSLConfiguration {

    /**
     * SSL Producer Factory with comprehensive SSL configuration
     */
    @Bean
    public ProducerFactory<String, Object> sslProducerFactory() {
        Map<String, Object> props = new HashMap<>();

        // Basic Kafka configuration
        props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9093"); //
SSL port
        props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
StringSerializer.class);
        props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
JsonSerializer.class);

        // CRITICAL: Enable SSL protocol
        props.put(CommonClientConfigs.SECURITY_PROTOCOL_CONFIG, "SSL");

        // SSL Truststore configuration (validates server certificates)
        props.put(SslConfigs.SSL_TRUSTSTORE_LOCATION_CONFIG,
"/path/to/kafka.client.truststore.jks");
        props.put(SslConfigs.SSL_TRUSTSTORE_PASSWORD_CONFIG, "truststore-
password");
        props.put(SslConfigs.SSL_TRUSTSTORE_TYPE_CONFIG, "JKS"); // JKS, PKCS12,
PEM

        // SSL Keystore configuration (client certificate for mTLS)
        props.put(SslConfigs.SSL_KEYSTORE_LOCATION_CONFIG,
"/path/to/kafka.client.keystore.jks");
        props.put(SslConfigs.SSL_KEYSTORE_PASSWORD_CONFIG, "keystore-password");
        props.put(SslConfigs.SSL_KEYSTORE_TYPE_CONFIG, "JKS");
        props.put(SslConfigs.SSL_KEY_PASSWORD_CONFIG, "key-password");

        // SSL Protocol and Cipher Configuration
        props.put(SslConfigs.SSL_PROTOCOL_CONFIG, "TLSv1.3"); // Use latest TLS
version
        props.put(SslConfigs.SSL_ENABLED_PROTOCOLS_CONFIG, "TLSv1.3,TLSv1.2");
        props.put(SslConfigs.SSL_CIPHER_SUITES_CONFIG,

"TLS_AES_256_GCM_SHA384,TLS_CHACHA20_POLY1305_SHA256,TLS_AES_128_GCM_SHA256");

        // SSL Endpoint Identification (hostname verification)
        props.put(SslConfigs.SSL_ENDPOINT_IDENTIFICATION_ALGORITHM_CONFIG,
"https");

        // SSL Provider (optional)
        props.put(SslConfigs.SSL_PROVIDER_CONFIG, ""); // Default provider
```

```
        // Performance optimization for SSL
        props.put(ProducerConfig.BATCH_SIZE_CONFIG, 65536); // 64KB batches
        props.put(ProducerConfig.LINGER_MS_CONFIG, 10);
        props.put(ProducerConfig.COMPRESSION_TYPE_CONFIG, "snappy");

        log.info("Configured SSL producer factory with comprehensive SSL
settings");

        return new DefaultKafkaProducerFactory<>(props);
    }

    /**
     * SSL Consumer Factory with comprehensive SSL configuration
     */
    @Bean
    public ConsumerFactory<String, Object> sslConsumerFactory() {
        Map<String, Object> props = new HashMap<>();

        // Basic Kafka configuration
        props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9093");
        props.put(ConsumerConfig.GROUP_ID_CONFIG, "ssl-consumer-group");
        props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class);
        props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
JsonDeserializer.class);

        // CRITICAL: Enable SSL protocol
        props.put(CommonClientConfigs.SECURITY_PROTOCOL_CONFIG, "SSL");

        // SSL Truststore configuration (same as producer)
        props.put(SslConfigs.SSL_TRUSTSTORE_LOCATION_CONFIG,
"/path/to/kafka.client.truststore.jks");
        props.put(SslConfigs.SSL_TRUSTSTORE_PASSWORD_CONFIG, "truststore-
password");
        props.put(SslConfigs.SSL_TRUSTSTORE_TYPE_CONFIG, "JKS");

        // SSL Keystore configuration (same as producer)
        props.put(SslConfigs.SSL_KEYSTORE_LOCATION_CONFIG,
"/path/to/kafka.client.keystore.jks");
        props.put(SslConfigs.SSL_KEYSTORE_PASSWORD_CONFIG, "keystore-password");
        props.put(SslConfigs.SSL_KEYSTORE_TYPE_CONFIG, "JKS");
        props.put(SslConfigs.SSL_KEY_PASSWORD_CONFIG, "key-password");

        // SSL Protocol configuration (same as producer)
        props.put(SslConfigs.SSL_PROTOCOL_CONFIG, "TLSv1.3");
        props.put(SslConfigs.SSL_ENABLED_PROTOCOLS_CONFIG, "TLSv1.3,TLSv1.2");
        props.put(SslConfigs.SSL_ENDPOINT_IDENTIFICATION_ALGORITHM_CONFIG,
"https");

        // Consumer-specific SSL optimizations
        props.put(ConsumerConfig.FETCH_MIN_BYTES_CONFIG, 50 * 1024); // 50KB
        props.put(ConsumerConfig.FETCH_MAX_WAIT_MS_CONFIG, 500);
```

```
        log.info("Configured SSL consumer factory with comprehensive SSL
settings");

        return new DefaultKafkaConsumerFactory<>(props);
    }

    /**
     * SSL KafkaTemplate
     */
    @Bean
    public KafkaTemplate<String, Object> sslKafkaTemplate() {
        KafkaTemplate<String, Object> template = new KafkaTemplate<>
(sslProducerFactory());

        // Configure default topic and error handling
        template.setDefaultTopic("default-ssl-topic");

        log.info("Created SSL-enabled KafkaTemplate");

        return template;
    }

    /**
     * SSL Listener Container Factory
     */
    @Bean
    public ConcurrentKafkaListenerContainerFactory<String, Object>
sslKafkaListenerContainerFactory() {
        ConcurrentKafkaListenerContainerFactory<String, Object> factory =
            new ConcurrentKafkaListenerContainerFactory<>();

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

        // Container properties for SSL

factory.getContainerProperties().setAckMode(ContainerProperties.AckMode.RECORD);
        factory.getContainerProperties().setSyncCommits(true);

        log.info("Configured SSL listener container factory");

        return factory;
    }
}

/**
 * Advanced SSL configuration with custom SSL context
 */
@Configuration
@lombok.extern.slf4j.Slf4j
public class AdvancedSSLConfiguration {

    /**
     * Custom SSL configuration with programmatic truststore/keystore setup
```

```
     */
    @Bean
    public ProducerFactory<String, Object> customSSLProducerFactory() {
        Map<String, Object> props = new HashMap<>();

        // Basic configuration
        props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9093");
        props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
StringSerializer.class);
        props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
JsonSerializer.class);

        // Enable SSL
        props.put(CommonClientConfigs.SECURITY_PROTOCOL_CONFIG, "SSL");

        // Custom SSL configuration
        try {
            // Load custom SSL context
            SSLContext sslContext = createCustomSSLContext();

            // Configure SSL with custom context
            props.put(SslConfigs.SSL_CONTEXT_CONFIG, sslContext);

        } catch (Exception e) {
            log.error("Failed to create custom SSL context", e);
            throw new RuntimeException("SSL configuration failed", e);
        }

        log.info("Configured custom SSL producer factory");

        return new DefaultKafkaProducerFactory<>(props);
    }

    /**
     * Create custom SSL context with programmatic certificate loading
     */
    private SSLContext createCustomSSLContext() throws Exception {

        // Load keystore programmatically
        KeyStore keystore = KeyStore.getInstance("JKS");
        try (FileInputStream keystoreFile = new
FileInputStream("/path/to/kafka.client.keystore.jks")) {
            keystore.load(keystoreFile, "keystore-password".toCharArray());
        }

        // Load truststore programmatically
        KeyStore truststore = KeyStore.getInstance("JKS");
        try (FileInputStream truststoreFile = new
FileInputStream("/path/to/kafka.client.truststore.jks")) {
            truststore.load(truststoreFile, "truststore-password".toCharArray());
        }

        // Initialize KeyManagerFactory
        KeyManagerFactory keyManagerFactory =
```

```
KeyManagerFactory.getInstance(KeyManagerFactory.getDefaultAlgorithm());
        keyManagerFactory.init(keystore, "key-password".toCharArray());

        // Initialize TrustManagerFactory
        TrustManagerFactory trustManagerFactory =
TrustManagerFactory.getInstance(TrustManagerFactory.getDefaultAlgorithm());
        trustManagerFactory.init(truststore);

        // Create SSL context
        SSLContext sslContext = SSLContext.getInstance("TLSv1.3");
        sslContext.init(
            keyManagerFactory.getKeyManagers(),
            trustManagerFactory.getTrustManagers(),
            new SecureRandom()
        );

        log.info("Created custom SSL context with programmatic certificate
loading");

        return sslContext;
    }

    /**
     * SSL configuration with environment-based certificate paths
     */
    @Bean
    public ConsumerFactory<String, Object> environmentSSLConsumerFactory() {
        Map<String, Object> props = new HashMap<>();

        // Basic configuration
        props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG,
            System.getenv().getOrDefault("KAFKA_BOOTSTRAP_SERVERS",
"localhost:9093"));
        props.put(ConsumerConfig.GROUP_ID_CONFIG, "env-ssl-consumer-group");
        props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class);
        props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
JsonDeserializer.class);

        // Enable SSL
        props.put(CommonClientConfigs.SECURITY_PROTOCOL_CONFIG, "SSL");

        // Environment-based SSL configuration
        String truststorePath =
System.getenv().getOrDefault("KAFKA_SSL_TRUSTSTORE_LOCATION",
            "/opt/kafka/ssl/kafka.client.truststore.jks");
        String truststorePassword =
System.getenv().getOrDefault("KAFKA_SSL_TRUSTSTORE_PASSWORD",
            "changeit");
        String keystorePath =
System.getenv().getOrDefault("KAFKA_SSL_KEYSTORE_LOCATION",
            "/opt/kafka/ssl/kafka.client.keystore.jks");
        String keystorePassword =
System.getenv().getOrDefault("KAFKA_SSL_KEYSTORE_PASSWORD",
```

```
            "changeit");
        String keyPassword =
System.getenv().getOrDefault("KAFKA_SSL_KEY_PASSWORD",
            "changeit");

        // SSL configuration from environment
        props.put(SslConfigs.SSL_TRUSTSTORE_LOCATION_CONFIG, truststorePath);
        props.put(SslConfigs.SSL_TRUSTSTORE_PASSWORD_CONFIG, truststorePassword);
        props.put(SslConfigs.SSL_TRUSTSTORE_TYPE_CONFIG, "JKS");

        props.put(SslConfigs.SSL_KEYSTORE_LOCATION_CONFIG, keystorePath);
        props.put(SslConfigs.SSL_KEYSTORE_PASSWORD_CONFIG, keystorePassword);
        props.put(SslConfigs.SSL_KEYSTORE_TYPE_CONFIG, "JKS");
        props.put(SslConfigs.SSL_KEY_PASSWORD_CONFIG, keyPassword);

        // Advanced SSL settings
        props.put(SslConfigs.SSL_PROTOCOL_CONFIG, "TLSv1.3");
        props.put(SslConfigs.SSL_ENABLED_PROTOCOLS_CONFIG, "TLSv1.3,TLSv1.2");

        // Optional: Disable hostname verification for development
        String disableHostnameVerification =
System.getenv("KAFKA_SSL_DISABLE_HOSTNAME_VERIFICATION");
        if ("true".equals(disableHostnameVerification)) {
            props.put(SslConfigs.SSL_ENDPOINT_IDENTIFICATION_ALGORITHM_CONFIG,
"");
            log.warn("SSL hostname verification disabled - not recommended for
production");
        } else {
            props.put(SslConfigs.SSL_ENDPOINT_IDENTIFICATION_ALGORITHM_CONFIG,
"https");
        }

        log.info("Configured environment-based SSL consumer factory: truststore=
{}, keystore={}",
            truststorePath, keystorePath);

        return new DefaultKafkaConsumerFactory<>(props);
    }
}

/**
 * SSL-enabled message producer service
 */
@Service
@lombok.extern.slf4j.Slf4j
public class SecureMessageProducer {

    @Autowired
    private KafkaTemplate<String, Object> sslKafkaTemplate;

    /**
     * Send secure message using SSL-encrypted connection
     */
    public void sendSecureMessage(String topic, String key, Object message) {
```

```
        log.info("Sending secure message over SSL: topic={}, key={}", topic, key);

        try {
            ListenableFuture<SendResult<String, Object>> future =
                sslKafkaTemplate.send(topic, key, message);

            future.addCallback(
                result -> {
                    RecordMetadata metadata = result.getRecordMetadata();
                    log.info("Secure message sent successfully: topic={},
partition={}, offset={}, timestamp={}",
                        metadata.topic(), metadata.partition(), metadata.offset(),
metadata.timestamp());
                },
                failure -> {
                    log.error("Failed to send secure message: topic={}, key={}",
topic, key, failure);
                }
            );

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

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

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

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

            for (KeyValuePair kvp : messages) {
                ListenableFuture<SendResult<String, Object>> future =
                    sslKafkaTemplate.send(topic, kvp.getKey(), kvp.getValue());
                futures.add(future);
            }

            // Wait for all messages to complete
            for (ListenableFuture<SendResult<String, Object>> future : futures) {
                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());

        } catch (Exception e) {
            log.error("Error sending secure message batch", e);
            throw e;
        }
    }
}

/**
 * SSL-enabled message consumer service
 */
@Component
@lombok.extern.slf4j.Slf4j
public class SecureMessageConsumer {

    /**
     * Consume messages over SSL-encrypted connection
     */
    @KafkaListener(
        topics = "secure-topic",
        groupId = "secure-consumer-group",
        containerFactory = "sslKafkaListenerContainerFactory"
    )
    public void consumeSecureMessage(@Payload String message,
                                     @Header(KafkaHeaders.RECEIVED_TOPIC) String
topic,
                                     @Header(KafkaHeaders.RECEIVED_PARTITION) int
partition,
                                     @Header(KafkaHeaders.OFFSET) long offset) {

        log.info("Received secure message over SSL: topic={}, partition={},
offset={}, message={}",
            topic, partition, offset, message);

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

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

    /**
     * Consume messages with SSL and manual acknowledgment
     */
```

```
    @KafkaListener(
        topics = "secure-manual-ack-topic",
        groupId = "secure-manual-ack-group",
        containerFactory = "sslKafkaListenerContainerFactory"
    )
    public void consumeSecureMessageWithAck(@Payload String message,
                                            @Header(KafkaHeaders.RECEIVED_TOPIC)
String topic,
                                            @Header(KafkaHeaders.OFFSET) long
offset,
                                            Acknowledgment ack) {

        log.info("Processing secure message with manual ack: topic={}, offset={}",
topic, offset);

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

            // Manual acknowledgment
            ack.acknowledge();

            log.info("Secure message processed and acknowledged: topic={}, offset=
{}", topic, offset);

        } catch (Exception e) {
            log.error("Error processing secure message - not acknowledging: topic=
{}, offset={}",
                topic, offset, e);
            throw e;
        }
    }

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

// Supporting data structures
@lombok.Data
@lombok.AllArgsConstructor
class KeyValuePair {
    private String key;
    private Object value;
}
```

Creating SSL Certificates and Keystores

```
#!/bin/bash
# Complete SSL certificate generation script for Kafka
```

```
# Set variables
VALIDITY_DAYS=365
KEYSTORE_PASSWORD="changeit"
TRUSTSTORE_PASSWORD="changeit"
KEY_PASSWORD="changeit"
CN="localhost"  # Common Name
OU="IT Department"  # Organizational Unit
O="My Company"  # Organization
L="San Francisco"  # Locality
ST="CA"  # State
C="US"  # Country

# Create certificate authority (CA)
echo "Creating Certificate Authority..."
openssl req -new -x509 -keyout ca-key -out ca-cert -days $VALIDITY_DAYS -subj
"/CN=kafka-ca/OU=$OU/O=$O/L=$L/ST=$ST/C=$C" -passout pass:$KEY_PASSWORD

# Create server keystore and key
echo "Creating server keystore..."
keytool -keystore kafka.server.keystore.jks -alias server -validity $VALIDITY_DAYS
-genkey -keyalg RSA \
  -dname "CN=$CN,OU=$OU,O=$O,L=$L,ST=$ST,C=$C" \
  -storepass $KEYSTORE_PASSWORD -keypass $KEY_PASSWORD

# Create certificate signing request for server
echo "Creating server certificate signing request..."
keytool -keystore kafka.server.keystore.jks -alias server -certreq -file server-
cert-request \
  -storepass $KEYSTORE_PASSWORD -keypass $KEY_PASSWORD

# Sign server certificate with CA
echo "Signing server certificate..."
openssl x509 -req -CA ca-cert -CAkey ca-key -in server-cert-request -out server-
cert-signed \
  -days $VALIDITY_DAYS -CAcreateserial -passin pass:$KEY_PASSWORD

# Import CA certificate into server keystore
echo "Importing CA certificate into server keystore..."
keytool -keystore kafka.server.keystore.jks -alias CARoot -import -file ca-cert \
  -storepass $KEYSTORE_PASSWORD -noprompt

# Import signed certificate into server keystore
echo "Importing signed certificate into server keystore..."
keytool -keystore kafka.server.keystore.jks -alias server -import -file server-
cert-signed \
  -storepass $KEYSTORE_PASSWORD -keypass $KEY_PASSWORD -noprompt

# Create server truststore
echo "Creating server truststore..."
keytool -keystore kafka.server.truststore.jks -alias CARoot -import -file ca-cert
\
  -storepass $TRUSTSTORE_PASSWORD -noprompt
```

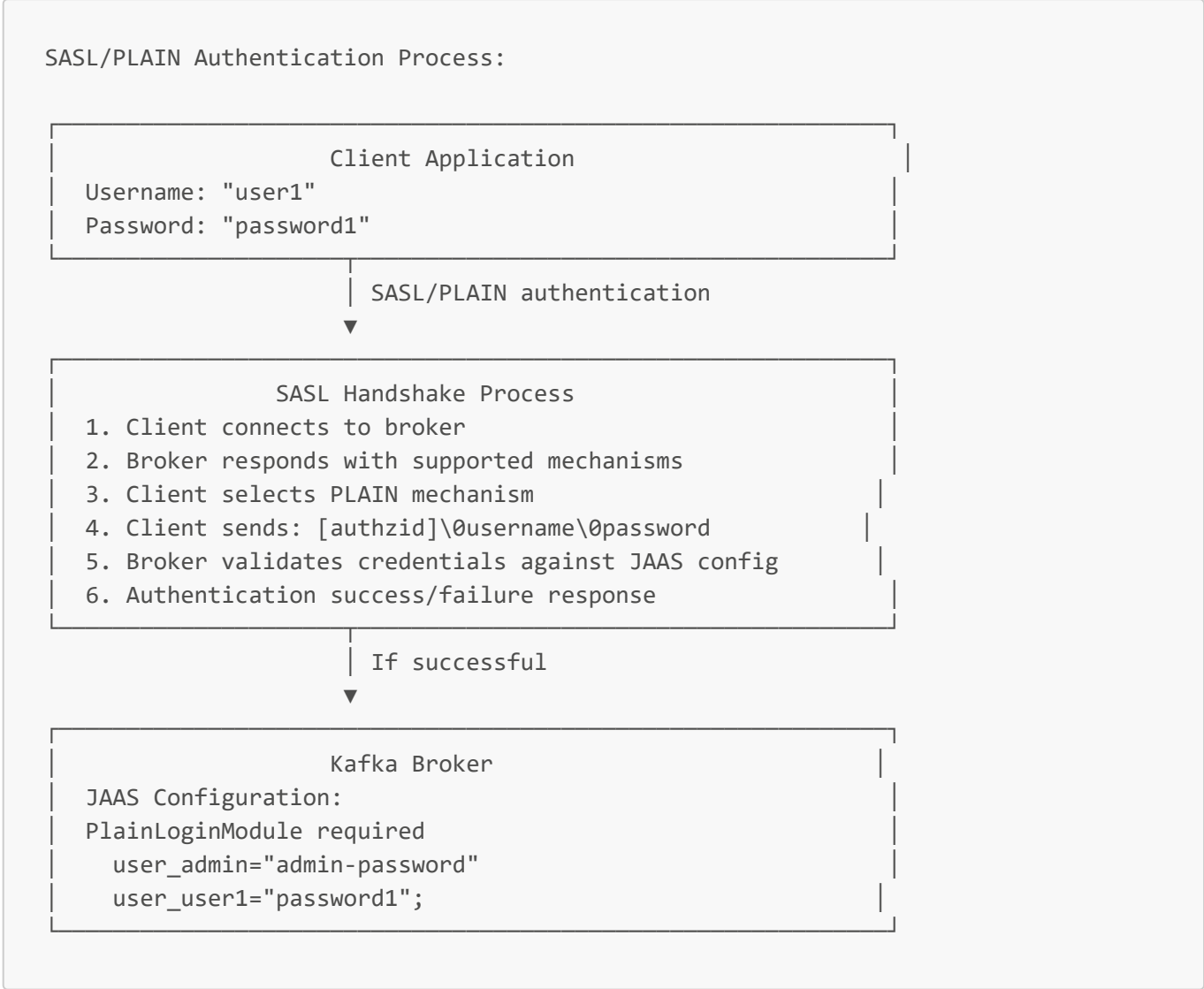
```
# Create client keystore and key
echo "Creating client keystore..."
keytool -keystore kafka.client.keystore.jks -alias client -validity $VALIDITY_DAYS
-genkey -keyalg RSA \
  -dname "CN=kafka-client,OU=$OU,O=$O,L=$L,ST=$ST,C=$C" \
  -storepass $KEYSTORE_PASSWORD -keypass $KEY_PASSWORD

# Create certificate signing request for client
echo "Creating client certificate signing request..."
keytool -keystore kafka.client.keystore.jks -alias client -certreq -file client-
cert-request \
  -storepass $KEYSTORE_PASSWORD -keypass $KEY_PASSWORD

# Sign client certificate with CA
echo "Signing client certificate..."
openssl x509 -req -CA ca-cert -CAkey ca-key -in client-cert-request -out client-
cert-signed \
  -days $VALIDITY_DAYS -CAcreateserial -passin pass:$KEY_PASSWORD

# Import CA certificate into client keystore
echo "Importing CA certificate into client keystore..."
keytool -keystore kafka.client.keystore.jks -alias CARoot -import -file ca-cert \
  -storepass $KEYSTORE_PASSWORD -noprompt

# Import signed certificate into client keystore
echo "Importing signed certificate into client keystore..."
keytool -keystore kafka.client.keystore.jks -alias client -import -file client-
cert-signed \
  -storepass $KEYSTORE_PASSWORD -keypass $KEY_PASSWORD -noprompt

# Create client truststore
echo "Creating client truststore..."
keytool -keystore kafka.client.truststore.jks -alias CARoot -import -file ca-cert
\
  -storepass $TRUSTSTORE_PASSWORD -noprompt

echo "SSL certificate generation completed!"
echo "Files generated:"
echo "  - ca-cert (CA certificate)"
echo "  - kafka.server.keystore.jks (server keystore)"
echo "  - kafka.server.truststore.jks (server truststore)"
echo "  - kafka.client.keystore.jks (client keystore)"
echo "  - kafka.client.truststore.jks (client truststore)"

# Cleanup temporary files
rm -f ca-key server-cert-request server-cert-signed client-cert-request client-
cert-signed

echo "Temporary files cleaned up."
```

SASL Authentication

SASL/PLAIN

Simple Explanation: SASL/PLAIN is a simple username/password authentication mechanism. Credentials are sent as plaintext over the connection, so it **must** be combined with SSL/TLS encryption in production environments to protect credentials during transmission.

SASL/PLAIN Authentication Flow:



Complete SASL/PLAIN Configuration

```
import org.apache.kafka.clients.CommonClientConfigs;
import org.apache.kafka.common.config.SaslConfigs;
import org.apache.kafka.common.security.plain.PlainLoginModule;

/**
 * Comprehensive SASL/PLAIN configuration for Spring Kafka
 */
@Configuration
@lombok.extern.slf4j.Slf4j
public class KafkaSASLPlainConfiguration {

    @Value("${kafka.sasl.username}")
```

```
    private String saslUsername;

    @Value("${kafka.sasl.password}")
    private String saslPassword;

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

    /**
     * SASL/PLAIN Producer Factory with SSL encryption
     */
    @Bean
    public ProducerFactory<String, Object> saslPlainProducerFactory() {
        Map<String, Object> props = new HashMap<>();

        // Basic Kafka configuration
        props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, bootstrapServers);
        props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
StringSerializer.class);
        props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
JsonSerializer.class);

        // CRITICAL: Use SASL_SSL for production (encrypts PLAIN credentials)
        props.put(CommonClientConfigs.SECURITY_PROTOCOL_CONFIG, "SASL_SSL");

        // SASL mechanism configuration
        props.put(SaslConfigs.SASL_MECHANISM, "PLAIN");

        // SASL JAAS configuration for PLAIN authentication
        String jaasConfig = String.format(
            "%s required username=\"%s\" password=\"%s\";",
            PlainLoginModule.class.getName(),
            saslUsername,
            saslPassword
        );
        props.put(SaslConfigs.SASL_JAAS_CONFIG, jaasConfig);

        // SSL configuration (required when using SASL_SSL)
        props.put(SslConfigs.SSL_TRUSTSTORE_LOCATION_CONFIG,
"/path/to/kafka.client.truststore.jks");
        props.put(SslConfigs.SSL_TRUSTSTORE_PASSWORD_CONFIG, "truststore-
password");
        props.put(SslConfigs.SSL_ENDPOINT_IDENTIFICATION_ALGORITHM_CONFIG,
"https");

        // Performance optimization
        props.put(ProducerConfig.ACKS_CONFIG, "all");
        props.put(ProducerConfig.RETRIES_CONFIG, Integer.MAX_VALUE);
        props.put(ProducerConfig.BATCH_SIZE_CONFIG, 32768);
        props.put(ProducerConfig.LINGER_MS_CONFIG, 5);

        log.info("Configured SASL/PLAIN producer factory: username={},
protocol=SASL_SSL", saslUsername);
```

```
        return new DefaultKafkaProducerFactory<>(props);
    }

    /**
     * SASL/PLAIN Consumer Factory with SSL encryption
     */
    @Bean
    public ConsumerFactory<String, Object> saslPlainConsumerFactory() {
        Map<String, Object> props = new HashMap<>();

        // Basic Kafka configuration
        props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, bootstrapServers);
        props.put(ConsumerConfig.GROUP_ID_CONFIG, "sasl-plain-consumer-group");
        props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class);
        props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
JsonDeserializer.class);

        // SASL_SSL protocol for secure PLAIN authentication
        props.put(CommonClientConfigs.SECURITY_PROTOCOL_CONFIG, "SASL_SSL");
        props.put(SaslConfigs.SASL_MECHANISM, "PLAIN");

        // SASL JAAS configuration
        String jaasConfig = String.format(
            "%s required username=\"%s\" password=\"%s\";",
            PlainLoginModule.class.getName(),
            saslUsername,
            saslPassword
        );
        props.put(SaslConfigs.SASL_JAAS_CONFIG, jaasConfig);

        // SSL configuration
        props.put(SslConfigs.SSL_TRUSTSTORE_LOCATION_CONFIG,
"/path/to/kafka.client.truststore.jks");
        props.put(SslConfigs.SSL_TRUSTSTORE_PASSWORD_CONFIG, "truststore-
password");
        props.put(SslConfigs.SSL_ENDPOINT_IDENTIFICATION_ALGORITHM_CONFIG,
"https");

        // Consumer-specific settings
        props.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest");
        props.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, false);
        props.put(ConsumerConfig.SESSION_TIMEOUT_MS_CONFIG, 30000);
        props.put(ConsumerConfig.HEARTBEAT_INTERVAL_MS_CONFIG, 10000);

        log.info("Configured SASL/PLAIN consumer factory: username={},
protocol=SASL_SSL", saslUsername);

        return new DefaultKafkaConsumerFactory<>(props);
    }

    /**
     * SASL/PLAIN KafkaTemplate
     */
```

```
    @Bean
    public KafkaTemplate<String, Object> saslPlainKafkaTemplate() {
        return new KafkaTemplate<>(saslPlainProducerFactory());
    }

    /**
     * SASL/PLAIN Listener Container Factory
     */
    @Bean
    public ConcurrentKafkaListenerContainerFactory<String, Object>
saslPlainKafkaListenerContainerFactory() {
        ConcurrentKafkaListenerContainerFactory<String, Object> factory =
            new ConcurrentKafkaListenerContainerFactory<>();

        factory.setConsumerFactory(saslPlainConsumerFactory());
        factory.setConcurrency(2);

        // Container properties

factory.getContainerProperties().setAckMode(ContainerProperties.AckMode.MANUAL_IMM
EDIATE);

        return factory;
    }

    /**
     * Development-only SASL/PLAIN without SSL (SASL_PLAINTEXT)
     * WARNING: Only use for development - credentials are sent in plaintext!
     */
    @Bean
    @Profile("dev")
    public ProducerFactory<String, Object> devSaslPlainProducerFactory() {
        Map<String, Object> props = new HashMap<>();

        // Basic configuration
        props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092"); //
Non-SSL port
        props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
StringSerializer.class);
        props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
JsonSerializer.class);

        // WARNING: SASL_PLAINTEXT - credentials sent in plaintext!
        props.put(CommonClientConfigs.SECURITY_PROTOCOL_CONFIG, "SASL_PLAINTEXT");
        props.put(SaslConfigs.SASL_MECHANISM, "PLAIN");

        // SASL JAAS configuration
        String jaasConfig = String.format(
            "%s required username=\"%s\" password=\"%s\";",
            PlainLoginModule.class.getName(),
            saslUsername,
            saslPassword
        );
        props.put(SaslConfigs.SASL_JAAS_CONFIG, jaasConfig);
```

```
        log.warn("Configured DEVELOPMENT-ONLY SASL_PLAINTEXT producer -
credentials sent in plaintext!");

        return new DefaultKafkaProducerFactory<>(props);
    }
}

/**
 * Multi-user SASL/PLAIN configuration with different credentials per service
 */
@Configuration
@lombok.extern.slf4j.Slf4j
public class MultiUserSASLPlainConfiguration {

    /**
     * Producer service with dedicated credentials
     */
    @Bean("producerServiceSaslFactory")
    public ProducerFactory<String, Object> producerServiceSaslFactory() {
        return createSaslPlainProducerFactory("producer-service", "producer-
secret");
    }

    /**
     * Consumer service with dedicated credentials
     */
    @Bean("consumerServiceSaslFactory")
    public ConsumerFactory<String, Object> consumerServiceSaslFactory() {
        return createSaslPlainConsumerFactory("consumer-service", "consumer-
secret", "consumer-service-group");
    }

    /**
     * Admin service with elevated privileges
     */
    @Bean("adminServiceSaslFactory")
    public ProducerFactory<String, Object> adminServiceSaslFactory() {
        return createSaslPlainProducerFactory("admin-user", "admin-secret");
    }

    private ProducerFactory<String, Object> createSaslPlainProducerFactory(String
username, String password) {
        Map<String, Object> props = new HashMap<>();

        props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9093");
        props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
StringSerializer.class);
        props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
JsonSerializer.class);

        // SASL_SSL configuration
        props.put(CommonClientConfigs.SECURITY_PROTOCOL_CONFIG, "SASL_SSL");
        props.put(SaslConfigs.SASL_MECHANISM, "PLAIN");
```

```
        String jaasConfig = String.format(
            "%s required username=\"%s\" password=\"%s\";",
            PlainLoginModule.class.getName(),
            username,
            password
        );
        props.put(SaslConfigs.SASL_JAAS_CONFIG, jaasConfig);

        // SSL truststore
        props.put(SslConfigs.SSL_TRUSTSTORE_LOCATION_CONFIG,
"/path/to/kafka.client.truststore.jks");
        props.put(SslConfigs.SSL_TRUSTSTORE_PASSWORD_CONFIG, "truststore-
password");

        log.info("Created SASL/PLAIN producer factory for user: {}", username);

        return new DefaultKafkaProducerFactory<>(props);
    }

    private ConsumerFactory<String, Object> createSaslPlainConsumerFactory(String
username, String password, String groupId) {
        Map<String, Object> props = new HashMap<>();

        props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9093");
        props.put(ConsumerConfig.GROUP_ID_CONFIG, groupId);
        props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class);
        props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
JsonDeserializer.class);

        // SASL_SSL configuration
        props.put(CommonClientConfigs.SECURITY_PROTOCOL_CONFIG, "SASL_SSL");
        props.put(SaslConfigs.SASL_MECHANISM, "PLAIN");

        String jaasConfig = String.format(
            "%s required username=\"%s\" password=\"%s\";",
            PlainLoginModule.class.getName(),
            username,
            password
        );
        props.put(SaslConfigs.SASL_JAAS_CONFIG, jaasConfig);

        // SSL truststore
        props.put(SslConfigs.SSL_TRUSTSTORE_LOCATION_CONFIG,
"/path/to/kafka.client.truststore.jks");
        props.put(SslConfigs.SSL_TRUSTSTORE_PASSWORD_CONFIG, "truststore-
password");

        // Consumer settings
        props.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest");
        props.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, false);

        log.info("Created SASL/PLAIN consumer factory for user: {}, group: {}",
```

```
username, groupId);

        return new DefaultKafkaConsumerFactory<>(props);
    }
}

/**
 * SASL/PLAIN service with authentication
 */
@Service
@lombok.extern.slf4j.Slf4j
public class AuthenticatedMessageService {

    @Autowired
    @Qualifier("producerServiceSaslFactory")
    private ProducerFactory<String, Object> producerFactory;

    private KafkaTemplate<String, Object> kafkaTemplate;

    @PostConstruct
    public void initKafkaTemplate() {
        this.kafkaTemplate = new KafkaTemplate<>(producerFactory);
    }

    /**
     * Send authenticated message using SASL/PLAIN
     */
    public void sendAuthenticatedMessage(String topic, String key, Object message)
{

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

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

            future.addCallback(
                result -> log.info("Authenticated message sent: offset={}",
                    result.getRecordMetadata().offset()),
                failure -> log.error("Failed to send authenticated message",
failure)
            );

        } catch (Exception e) {
            log.error("Error sending authenticated message", e);
            throw e;
        }
    }
}

/**
 * SASL/PLAIN authenticated consumer
 */
@Component
```

```
@lombok.extern.slf4j.Slf4j
public class AuthenticatedMessageConsumer {

    /**
     * Consume authenticated messages using SASL/PLAIN
     */
    @KafkaListener(
        topics = "authenticated-topic",
        groupId = "authenticated-consumer-group",
        containerFactory = "saslPlainKafkaListenerContainerFactory"
    )
    public void consumeAuthenticatedMessage(@Payload String message,
                                            @Header(KafkaHeaders.RECEIVED_TOPIC)
String topic,
                                            @Header(KafkaHeaders.OFFSET) long
offset,
                                            Acknowledgment ack) {

        log.info("Received authenticated message: topic={}, offset={}, message=
{}",
            topic, offset, message);

        try {
            // Process authenticated message
            processAuthenticatedMessage(message);

            // Manual acknowledgment
            ack.acknowledge();

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

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

SASL/SCRAM

Simple Explanation: SASL/SCRAM (Salted Challenge Response Authentication Mechanism) is a more secure authentication method that doesn't send passwords over the network. Instead, it uses a challenge-response mechanism with salted password hashes stored in ZooKeeper, providing better security than SASL/PLAIN.

SASL/SCRAM Authentication Flow:

```
SASL/SCRAM Authentication Process:

┌─────────────────────────────────────────────────────────────┐
│                    Client Application                       │
│  Username: "user1"                                          │
│  Password: "password1" (never sent over network)            │
└─────────────────────┬───────────────────────────────────────┘
                      │ SASL/SCRAM challenge-response
                      ▼
┌─────────────────────────────────────────────────────────────┐
│              SASL/SCRAM Handshake Process                   │
│  1. Client → Broker: Username + random nonce                │
│  2. Broker → Client: Salt + iteration count + server nonce  │
│  3. Client computes: SaltedPassword = Hi(password, salt)    │
│  4. Client → Broker: Client proof (derived from password)   │
│  5. Broker validates proof against stored credentials       │
│  6. Broker → Client: Server signature for verification      │
│  7. Client validates server signature                       │
└─────────────────────┬───────────────────────────────────────┘
                      │ If successful
                      ▼
┌─────────────────────────────────────────────────────────────┐
│                   ZooKeeper/KRaft                           │
│  Stored SCRAM credentials:                                  │
│  /config/users/user1:                                       │
│    SCRAM-SHA-256=[salt,iterations,storedkey,serverkey]      │
│  Password never stored in plaintext                         │
└─────────────────────────────────────────────────────────────┘
```

Complete SASL/SCRAM Configuration

```
import org.apache.kafka.common.security.scram.ScramLoginModule;

/**
 * Comprehensive SASL/SCRAM configuration for Spring Kafka
 */
@Configuration
@lombok.extern.slf4j.Slf4j
public class KafkaSASLScramConfiguration {

    @Value("${kafka.sasl.username}")
    private String saslUsername;

    @Value("${kafka.sasl.password}")
    private String saslPassword;

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

    /**
```

```
     * SASL/SCRAM-SHA-256 Producer Factory
     */
    @Bean
    public ProducerFactory<String, Object> saslScramProducerFactory() {
        Map<String, Object> props = new HashMap<>();

        // Basic Kafka configuration
        props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, bootstrapServers);
        props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
StringSerializer.class);
        props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
JsonSerializer.class);

        // SASL_SSL protocol for secure SCRAM authentication
        props.put(CommonClientConfigs.SECURITY_PROTOCOL_CONFIG, "SASL_SSL");

        // CRITICAL: SCRAM-SHA-256 mechanism (more secure than PLAIN)
        props.put(SaslConfigs.SASL_MECHANISM, "SCRAM-SHA-256");

        // SASL JAAS configuration for SCRAM
        String jaasConfig = String.format(
            "%s required username=\"%s\" password=\"%s\";",
            ScramLoginModule.class.getName(),
            saslUsername,
            saslPassword
        );
        props.put(SaslConfigs.SASL_JAAS_CONFIG, jaasConfig);

        // SSL configuration (required with SASL_SSL)
        props.put(SslConfigs.SSL_TRUSTSTORE_LOCATION_CONFIG,
"/path/to/kafka.client.truststore.jks");
        props.put(SslConfigs.SSL_TRUSTSTORE_PASSWORD_CONFIG, "truststore-
password");
        props.put(SslConfigs.SSL_ENDPOINT_IDENTIFICATION_ALGORITHM_CONFIG,
"https");

        // Performance configuration
        props.put(ProducerConfig.ACKS_CONFIG, "all");
        props.put(ProducerConfig.RETRIES_CONFIG, Integer.MAX_VALUE);
        props.put(ProducerConfig.MAX_IN_FLIGHT_REQUESTS_PER_CONNECTION, 1); // For
ordering
        props.put(ProducerConfig.BATCH_SIZE_CONFIG, 32768);
        props.put(ProducerConfig.LINGER_MS_CONFIG, 5);

        log.info("Configured SASL/SCRAM-SHA-256 producer factory: username={}",
saslUsername);

        return new DefaultKafkaProducerFactory<>(props);
    }

    /**
     * SASL/SCRAM-SHA-256 Consumer Factory
     */
    @Bean
```

```
    public ConsumerFactory<String, Object> saslScramConsumerFactory() {
        Map<String, Object> props = new HashMap<>();

        // Basic Kafka configuration
        props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, bootstrapServers);
        props.put(ConsumerConfig.GROUP_ID_CONFIG, "sasl-scram-consumer-group");
        props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class);
        props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
JsonDeserializer.class);

        // SASL_SSL with SCRAM-SHA-256
        props.put(CommonClientConfigs.SECURITY_PROTOCOL_CONFIG, "SASL_SSL");
        props.put(SaslConfigs.SASL_MECHANISM, "SCRAM-SHA-256");

        // SASL JAAS configuration
        String jaasConfig = String.format(
            "%s required username=\"%s\" password=\"%s\";",
            ScramLoginModule.class.getName(),
            saslUsername,
            saslPassword
        );
        props.put(SaslConfigs.SASL_JAAS_CONFIG, jaasConfig);

        // SSL configuration
        props.put(SslConfigs.SSL_TRUSTSTORE_LOCATION_CONFIG,
"/path/to/kafka.client.truststore.jks");
        props.put(SslConfigs.SSL_TRUSTSTORE_PASSWORD_CONFIG, "truststore-
password");
        props.put(SslConfigs.SSL_ENDPOINT_IDENTIFICATION_ALGORITHM_CONFIG,
"https");

        // Consumer-specific settings
        props.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest");
        props.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, false);
        props.put(ConsumerConfig.SESSION_TIMEOUT_MS_CONFIG, 30000);
        props.put(ConsumerConfig.HEARTBEAT_INTERVAL_MS_CONFIG, 10000);
        props.put(ConsumerConfig.MAX_POLL_RECORDS_CONFIG, 100);

        log.info("Configured SASL/SCRAM-SHA-256 consumer factory: username={}",
saslUsername);

        return new DefaultKafkaConsumerFactory<>(props);
    }

    /**
     * SASL/SCRAM-SHA-512 Producer Factory (stronger hashing)
     */
    @Bean("scramSha512ProducerFactory")
    public ProducerFactory<String, Object> saslScramSha512ProducerFactory() {
        Map<String, Object> props = new HashMap<>();

        // Basic configuration
        props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, bootstrapServers);
```

```
        props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
StringSerializer.class);
        props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
JsonSerializer.class);

        // SASL_SSL with SCRAM-SHA-512 (stronger than SHA-256)
        props.put(CommonClientConfigs.SECURITY_PROTOCOL_CONFIG, "SASL_SSL");
        props.put(SaslConfigs.SASL_MECHANISM, "SCRAM-SHA-512");

        // JAAS configuration for SCRAM-SHA-512
        String jaasConfig = String.format(
            "%s required username=\"%s\" password=\"%s\";",
            ScramLoginModule.class.getName(),
            saslUsername,
            saslPassword
        );
        props.put(SaslConfigs.SASL_JAAS_CONFIG, jaasConfig);

        // SSL configuration
        props.put(SslConfigs.SSL_TRUSTSTORE_LOCATION_CONFIG,
"/path/to/kafka.client.truststore.jks");
        props.put(SslConfigs.SSL_TRUSTSTORE_PASSWORD_CONFIG, "truststore-
password");

        log.info("Configured SASL/SCRAM-SHA-512 producer factory: username={}",
saslUsername);

        return new DefaultKafkaProducerFactory<>(props);
    }

    /**
     * SASL/SCRAM KafkaTemplate
     */
    @Bean
    public KafkaTemplate<String, Object> saslScramKafkaTemplate() {
        return new KafkaTemplate<>(saslScramProducerFactory());
    }

    /**
     * SASL/SCRAM Listener Container Factory
     */
    @Bean
    public ConcurrentKafkaListenerContainerFactory<String, Object>
saslScramKafkaListenerContainerFactory() {
        ConcurrentKafkaListenerContainerFactory<String, Object> factory =
            new ConcurrentKafkaListenerContainerFactory<>();

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

        // Container properties for SCRAM authentication

factory.getContainerProperties().setAckMode(ContainerProperties.AckMode.MANUAL_IMM
EDIATE);
```

```
        factory.getContainerProperties().setSyncCommits(true);

        return factory;
    }
}

/**
 * SCRAM credential management service
 */
@Service
@lombok.extern.slf4j.Slf4j
public class ScramCredentialManagementService {

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

    @Autowired
    private AdminClient adminClient;

    /**
     * Create SCRAM credentials programmatically
     */
    public void createScramUser(String username, String password, ScramMechanism
mechanism) {

        log.info("Creating SCRAM user: username={}, mechanism={}", username,
mechanism);

        try {
            // Create SCRAM credential alteration
            Map<String, String> configs = new HashMap<>();

            switch (mechanism) {
                case SCRAM_SHA_256:
                    configs.put("SCRAM-SHA-256", String.format("[password=%s]",
password));
                    break;
                case SCRAM_SHA_512:
                    configs.put("SCRAM-SHA-512", String.format("[password=%s]",
password));
                    break;
                default:
                    throw new IllegalArgumentException("Unsupported SCRAM
mechanism: " + mechanism);
            }

            // Create config resource for user
            ConfigResource userResource = new
ConfigResource(ConfigResource.Type.USER, username);

            // Create alter configs operation
            Map<ConfigResource, Collection<AlterConfigOp>> alterOps = new
HashMap<>();
            alterOps.put(userResource, configs.entrySet().stream()
```

```
                .map(entry -> new AlterConfigOp(
                    new ConfigEntry(entry.getKey(), entry.getValue()),
                    AlterConfigOp.OpType.SET))
                .collect(Collectors.toList()));

            // Execute the operation
            AlterConfigsResult result =
adminClient.incrementalAlterConfigs(alterOps);
            result.all().get(30, TimeUnit.SECONDS);

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

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

    /**
     * Delete SCRAM credentials
     */
    public void deleteScramUser(String username, ScramMechanism mechanism) {

        log.info("Deleting SCRAM user: username={}, mechanism={}", username,
mechanism);

        try {
            ConfigResource userResource = new
ConfigResource(ConfigResource.Type.USER, username);

            Map<ConfigResource, Collection<AlterConfigOp>> alterOps = new
HashMap<>();

            String mechanismKey = mechanism == ScramMechanism.SCRAM_SHA_256 ?
                "SCRAM-SHA-256" : "SCRAM-SHA-512";

            alterOps.put(userResource, Collections.singletonList(
                new AlterConfigOp(new ConfigEntry(mechanismKey, ""),
AlterConfigOp.OpType.DELETE)));

            AlterConfigsResult result =
adminClient.incrementalAlterConfigs(alterOps);
            result.all().get(30, TimeUnit.SECONDS);

            log.info("SCRAM user deleted successfully: username={}", username);

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

    /**
     * List SCRAM users
```

```
     */
    public Map<String, List<String>> listScramUsers() {

        log.info("Listing SCRAM users");

        try {
            ConfigResource userResource = new
ConfigResource(ConfigResource.Type.USER, null);
            DescribeConfigsResult result =
adminClient.describeConfigs(Collections.singleton(userResource));

            Map<ConfigResource, Config> configs = result.all().get(30,
TimeUnit.SECONDS);
            Map<String, List<String>> scramUsers = new HashMap<>();

            for (Map.Entry<ConfigResource, Config> entry : configs.entrySet()) {
                String username = entry.getKey().name();
                List<String> mechanisms = new ArrayList<>();

                for (ConfigEntry configEntry : entry.getValue().entries()) {
                    if (configEntry.name().startsWith("SCRAM-")) {
                        mechanisms.add(configEntry.name());
                    }
                }

                if (!mechanisms.isEmpty()) {
                    scramUsers.put(username, mechanisms);
                }
            }

            log.info("Found {} SCRAM users", scramUsers.size());

            return scramUsers;

        } catch (Exception e) {
            log.error("Failed to list SCRAM users", e);
            throw new RuntimeException("SCRAM user listing failed", e);
        }
    }

    public enum ScramMechanism {
        SCRAM_SHA_256,
        SCRAM_SHA_512
    }
}

/**
 * SCRAM-authenticated service
 */
@Service
@lombok.extern.slf4j.Slf4j
public class ScramAuthenticatedService {

    @Autowired
```

```
    private KafkaTemplate<String, Object> saslScramKafkaTemplate;

    /**
     * Send message using SCRAM authentication
     */
    public void sendScramAuthenticatedMessage(String topic, String key, Object
message) {

        log.info("Sending SCRAM-authenticated message: topic={}, key={}", topic,
key);

        try {
            ListenableFuture<SendResult<String, Object>> future =
                saslScramKafkaTemplate.send(topic, key, message);

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

        } catch (Exception e) {
            log.error("Error sending SCRAM-authenticated message", e);
            throw e;
        }
    }
}

/**
 * SCRAM-authenticated consumer
 */
@Component
@lombok.extern.slf4j.Slf4j
public class ScramAuthenticatedConsumer {

    /**
     * Consume SCRAM-authenticated messages
     */
    @KafkaListener(
        topics = "scram-topic",
        groupId = "scram-consumer-group",
        containerFactory = "saslScramKafkaListenerContainerFactory"
    )
    public void consumeScramMessage(@Payload String message,
                                    @Header(KafkaHeaders.RECEIVED_TOPIC) String
topic,
                                    @Header(KafkaHeaders.OFFSET) long offset,
                                    Acknowledgment ack) {

        log.info("Received SCRAM-authenticated message: topic={}, offset={},
message={}",
            topic, offset, message);

        try {
```

```
            // Process SCRAM-authenticated message
            processScramMessage(message);

            // Manual acknowledgment
            ack.acknowledge();

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

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

This completes Part 1 of the Spring Kafka Security guide, covering SSL/TLS setup and SASL authentication (both PLAIN and SCRAM). The guide continues with Spring Boot property-based configuration in the next part.