

Compte Rendu  
TP Intergiciel et programmation par composants



Réalisé par:

- BADI Rania
- GAYE Khady
- JALIDY Amina

Encadré par: TONDEUR Hervé

## INTRODUCTION

Dans le cadre de notre cours d'Intergiciel et Programmation par Composants, nous avons réalisé un TP sur la sécurisation d'un serveur Apache Kafka via le protocole SSL. Ce rapport, réalisé par BADI Rania, GAYE Khady et JALIDY Amina sous la supervision de TONDEUR Hervé détaille l'étude de la sécurité SSL appliquée à Kafka, sa mise en pratique, et une démonstration de notre serveur sécurisé.

### I- [ETUDE] : Présentation de la sécurité SSL appliquée à KAFKA

#### a) Quelles sont les principales caractéristiques de la sécurisation via SSL d'un broker Kafka ?

1. **Confidentialité:** SSL (Secure Sockets Layer) assure que les données transmises entre les clients (producteurs/consommateurs) et les brokers sont chiffrées, rendant difficile leur interception et lecture par des tiers non autorisés.

2. **Authentification:** SSL utilise des certificats pour authentifier les parties communicantes. Cela permet de s'assurer que les clients se connectent aux brokers légitimes et vice versa.

3. **Intégrité des données:** SSL garantit que les données transmises n'ont pas été altérées en transit. Toute modification des données sera détectée.

4. Configuration flexible: Kafka permet de configurer SSL indépendamment pour chaque composant, que ce soit entre producteurs, brokers et consumers.

#### b) Quel est l'apport de ce principe sur la sécurité d'accès à Kafka ?

- **Protection des données sensibles:** En chiffrant les données en transit, SSL protège contre les attaques de type "man-in-the-middle" et autres interceptions de données sensibles.

- **Contrôle d'accès renforcé:** L'authentification mutuelle via des certificats SSL assure que seuls les clients et brokers autorisés peuvent se connecter les uns aux autres, réduisant le risque d'accès non autorisé.

- **Prévention des attaques d'altération:** L'intégrité des messages est vérifiée, empêchant ainsi les attaques où des messages pourraient être interceptés et modifiés avant d'atteindre leur destination.

#### c) Expliquer la sécurisation Producer vers broker, broker vers consumer, broker vers broker et comment cela se passe dans un mode cluster

- **Producer vers Broker:** Le producteur utilise SSL pour établir une connexion sécurisée avec le broker. Les données envoyées sont chiffrées et le broker authentifie le producteur à l'aide de certificats.

- **Broker vers Consumer:** De même, le broker utilise SSL pour sécuriser les communications vers le consommateur. Les données sont chiffrées en transit et le consommateur est authentifié par le broker.

- **Broker vers Broker:** Dans un cluster Kafka, les brokers communiquent entre eux pour répliquer les données et coordonner. SSL sécurise ces communications internes, garantissant que seules les communications entre brokers authentiques sont possibles.

- **Mode Cluster:** En mode cluster, chaque broker utilise SSL pour ses communications avec les autres brokers, les producteurs et les consommateurs. Cela assure une sécurité bout en bout au sein du cluster, empêchant tout accès non autorisé aux données à n'importe quel point de transmission.

#### d) Qu'en est-il de Zookeeper, faut-il également le sécuriser ?

Lorsque l'on parle de ZooKeeper dans le cadre de Kafka, il est essentiel de sécuriser également ZooKeeper. En effet, ZooKeeper est crucial pour la coordination et la gestion des métadonnées de Kafka. Si la sécurité de ZooKeeper n'est pas adéquate, cela peut créer des vulnérabilités et compromettre la sécurité globale du cluster Kafka. Pour assurer la sécurité de ZooKeeper, il est impératif de prendre les mesures suivantes :

**Authentification SASL :** On configure ZooKeeper pour utiliser l'authentification SASL, ce qui permet une authentification forte des clients se connectant à ZooKeeper.

**Chiffrement SSL/TLS :** On active le chiffrement SSL/TLS pour chiffrer les communications entre les clients et ZooKeeper, empêchant ainsi l'interception non autorisée des données.

**Autorisations et contrôle d'accès :** On essaie de configurer des règles d'autorisation appropriées pour contrôler l'accès aux ressources de ZooKeeper. Limitez les priviléges aux seuls utilisateurs et services nécessaires.

**Sécurité du système d'exploitation :** On s'assure que le système d'exploitation sur lequel ZooKeeper s'exécute est sécurisé en appliquant les bonnes pratiques de sécurité, telles que les correctifs réguliers, la configuration appropriée des pare-feux, etc.

#### e) Peut-on sécuriser également un cluster KRAFT avec SSL ? comment ?

Kraft (Kafka Raft Metadata mode) est une nouvelle façon de gérer les métadonnées Kafka sans dépendre de Zookeeper. La sécurisation d'un cluster KRaft avec SSL suit des principes similaires à ceux d'un cluster Kafka traditionnel :

**1. Configuration SSL pour les brokers:** Chaque broker Kraft est configuré pour utiliser SSL pour toutes ses communications internes et externes. Cela inclut la communication entre les brokers ainsi qu'entre les brokers et les clients.

**2. Certificats et clés:** Les certificats SSL et les clés doivent être générés et distribués à chaque broker. Ils sont utilisés pour authentifier les communications entre les brokers et les clients.

**3. Configurations spécifiques:** Les propriétés SSL doivent être définies dans les fichiers de configuration du broker (par exp: `server.properties`), incluant les chemins des certificats, les mots de passe des clés, et les protocoles SSL à utiliser.

Exemple de configuration dans `server.properties`:

```
```properties
listeners=SSL://:9093
ssl.keystore.location=/path/to/keystore.jks
ssl.keystore.password=your_keystore_password
ssl.key.password=your_key_password
ssl.truststore.location=/path/to/truststore.jks
ssl.truststore.password=your_truststore_password
security.inter.broker.protocol=SSL
````
```

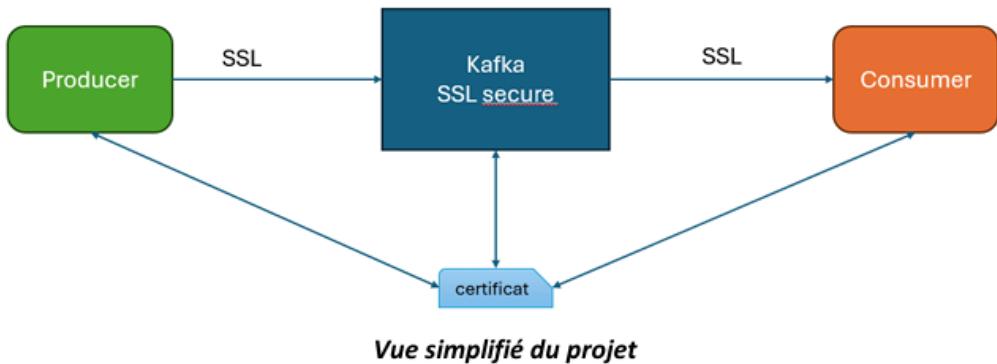
En configurant ainsi les brokers Kraft, on garantit que toutes les communications au sein du cluster sont sécurisées par SSL.

## II- [MISE EN PRATIQUE] : Implémentation de la sécurité via SSL à partir d'un projet:

Dans la version 0.9.0.0, la communauté Kafka a ajouté plusieurs fonctionnalités améliorant la sécurité des clusters Kafka. Les mesures de sécurité prises en charge incluent l'**authentification** des connexions aux brokers par les clients, autres brokers et outils via **SSL** ou **SASL** et l'authentification des connexions des brokers à ZooKeeper. De plus, les données transférées entre brokers et clients peuvent être cryptées via SSL, et l'autorisation des opérations de lecture/écriture par les clients est possible, avec support

pour l'intégration de services d'autorisation externes. La sécurité est optionnelle, permettant des configurations mixtes sécurisées et non sécurisées.

Le but du projet est de rajouter une **couche SSL** sur notre serveur Apache Kafka. Ainsi, les échanges de données entre le Producer Java Kafka et le broker, ainsi que la lecture des messages par le consommateur, seront protégés par le protocole SSL.



Le projet se divise en 2 phases essentielles : la première porte sur la sécurisation du serveur tandis que la seconde partie se concentre sur la sécurisation du client

- ❖ La sécurisation de la **partie serveur**
  - Génération des certificats
  - La configuration de server.properties
  
- ❖ La sécurisation de la **partie client**
  - Génération des certificats
  - La sécurisation des accès par le producteur
  - La sécurisation des accès par le consommateur
  - La configuration de docker-compose.yml

## 1. La sécurisation de la partie serveur

SSL (Secure Sockets Layer) est un protocole de sécurité conçu pour établir une connexion cryptée entre un client (comme un navigateur web) et un serveur (comme un serveur web). Cette connexion cryptée garantit que toutes les données transmises entre le client et le serveur restent privées et intactes. SSL utilise des certificats numériques pour authentifier les parties en communication et chiffrer les informations échangées, protégeant ainsi contre l'interception et les attaques. SSL a été largement remplacé par son successeur,

TLS (Transport Layer Security), qui offre des améliorations en termes de sécurité et de performance.

Avant de passer vers l'explication de notre déploiement, on va d'abord définir certains concepts fondamentales en SSL:

- ❖ Une **Autorité de certification** est une entité de confiance qui émet et valide les certificats numériques. Elle vérifie l'identité des organismes ou des individus et signe leurs certificats numériques, garantissant ainsi leur authenticité.
- ❖ Un **Keystore** : Un keystore est un référentiel qui stocke les clés privées et les certificats d'un serveur ou d'un client. Il est utilisé pour stocker et gérer les clés cryptographiques nécessaires à l'établissement de connexions SSL/TLS sécurisées. Un serveur utilise les certificats et les clés privées stockés dans le keystore pour prouver son identité aux clients.
- ❖ Un **Certificate signing request(CSR)**: est une demande envoyée à une CA pour obtenir un certificat numérique. Il contient des informations sur l'entité demandant le certificat, telles que le nom de domaine, l'organisation et la clé publique.
- ❖ Un **Truststore** est un référentiel qui contient les certificats d'autorité (CA) en lesquels un système a confiance. Lorsqu'un client reçoit un certificat d'un serveur, il vérifie le certificat par rapport aux certificats stockés dans le truststore. Si le certificat est signé par une CA présente dans le truststore, la connexion est considérée comme fiable.

La sécurisation de notre serveur en utilisant le protocole SSL par les étapes suivantes :

A. La mise en place d'un certificat d'autorité(CA) :

```
(khady㉿kali)-[~/kafka/bin]$ openssl req -new -newkey rsa:4096 -days 365 -x509 -subj "/CN=TPKafka" -keyout ca-key -out ca-cert -nodes
```

B. La création d'un certificat de serveur Kafka et son stockage dans le Keystore :

```
(khady㉿kali)-[~/kafka/bin]$ keytool -genkey -keystore kafka.server.keystore.jks -validity 365 -storepass pwdtpkafka -keypass pwdtpkafka -dname "CN=localhost" -storetype pkcs12 -keyalg RSA
```

C. La création d'un certificate signing request(CSR) : La CA utilise le CSR pour créer et signer un certificat numérique, liant la clé publique aux informations de l'entité.

```
(khady㉿kali)-[~/kafka/bin]$ keytool -keystore kafka.server.keystore.jks -certreq -file cert-file -storepass pwdtpkafka -keypass pwdtpkafka
```

- D. La signature du certificat utilisant le CA : La CA vérifie les informations dans le CSR et, si elles sont valides, utilise sa propre clé privée pour signer le CSR. Cette signature crée un certificat numérique qui lie la clé publique du demandeur à son identité.

```
(khady㉿kali)-[~/kafka/bin]$ openssl x509 -req -CA ca-cert.pem -CAkey ca-key.pem -in client-cert-file -out client-cert-signed -days 365 -CAcreateserial
```

- E. L'import du CA dans le Keystore : Le certificat signé par la CA, ainsi que la chaîne de certificats (qui peut inclure des certificats intermédiaires), sont importés dans le keystore de l'entité.

```
(khady㉿kali)-[~/kafka/bin]$ keytool -keystore kafka.server.keystore.jks -alias CARoot -import -file ca-cert.pem -storepass pwdtpkafka -keypass pwdtpkafka -noprompt
```

- F. L'import du CSR dans le Keystore :

```
(khady㉿kali)-[~/kafka/bin]$ keytool -keystore kafka.server.keystore.jks -import -file cert-file-signed -storepass pwdtpkafka -keypass pwdtpkafka -noprompt
```

- G. L'import du CA dans le Truststore : Le truststore doit contenir le certificat de la CA pour qu'il puisse être utilisé pour vérifier les certificats signés par cette CA.

```
(khady㉿kali)-[~/kafka/bin]$ keytool -keystore kafka.server.truststore.jks -alias CARoot -import -file ca-cert.pem -storepass pwdtpkafka -keypass pwdtpkafka -noprompt
```

Notre serveur dispose maintenant des certificats nécessaires pour communiquer de manière sécurisée. A présent nous allons configurer le fichier **server.properties** afin de rajouter les **chemins des stores** et les **clés de connexions**. Le fichier server.properties en serveur Apache est utilisé pour définir les **configurations essentielles** telles que les paramètres de connexion réseau, les options de sécurité et les réglages de performance pour assurer le bon fonctionnement du serveur.

Voici une capture du paragraphe rajouté dans ce fichier.

```
#####
# SSL Configurations #####
listeners=SSL://:9093
advertised.listeners=SSL://localhost:9093
ssl.keystore.location=/home/khady/sslDir/kafka.server.keystore.jks
ssl.keystore.password=pwdtpkafka
ssl.key.password=pwdtpkafka
ssl.truststore.location=/home/khady/sslDir/kafka.server.truststore.jks
ssl.truststore.password=pwdtpkafka
ssl.client.auth=required
security.inter.broker.protocol=SSL
```

Pour établir une connexion sécurisée via SSL/TLS, il est nécessaire de configurer SSL/TLS sur le client en plus du serveur, car le processus de handshake SSL/TLS implique une interaction entre les deux parties.

Le processus de handshake SSL/TLS est une série d'échanges entre le client et le serveur pour authentifier les deux parties, négocier les paramètres de cryptage, et établir des clés de session sécurisées pour chiffrer la communication.

## 2. La sécurisation de la partie client

On commence d'abord par la création des certificats ssl pour le client. On refait le même processus décrit avant pour le serveur mais là pour le client. On va utiliser la même autorité de certification générée dans la partie serveur.

### A. Création d'un keystore pour le client

```
(khady㉿kali)-[~/kafka/bin]
$ keytool -keystore kafka.client.keystore.jks -alias client -validity 365 -genkey -keyalg RSA
```

### B. Génération de clés publiques et clés privées pour le client

```
(khady㉿kali)-[~/kafka/bin]
$ keytool -genkey -keystore kafka.client.keystore.jks -alias client -validity 365 -storepass pwdtpkafka -keypass pwdtpkafka -dname "CN=localhost" -storetype pkcs12
```

### C. Importation du certificat de l'autorité dans le trustore du client

```
(khady㉿kali)-[~/kafka/bin]
└─$ keytool -keystore kafka.client.truststore.jks -alias CARoot -import -file ca-cert.pem
```

### D. Importation des certificats dans le keystore du client

```
(khady㉿kali)-[~/kafka/bin]
└─$ keytool -keystore kafka.client.keystore.jks -alias CARoot -import -file ca-cert.pem
```

```
(khady㉿kali)-[~/kafka/bin]
└─$ keytool -keystore kafka.client.keystore.jks -alias client -certreq -file client-cert-file
```

### E. Signature du certificat par le CA

```
(khady㉿kali)-[~/kafka/bin]
└─$ openssl x509 -req -CA ca-cert.pem -CAkey ca-key.pem -in client-cert-file -out client-cert-signed -days 365 -CAcreateserial
```

### F. Importation des certificats signés dans le keystore

```
(khady㉿kali)-[~/kafka/bin]
└─$ keytool -keystore kafka.client.truststore.jks -alias CARoot -import -file ca-cert.pem
```

```
(khady㉿kali)-[~/kafka/bin]
└─$ keytool -keystore kafka.client.keystore.jks -alias client -import -file client-cert-signed
```

Maintenant la configuration des certificats ssl dans la partie est terminée, on va modifier les fichiers du projet pour permettre une connexion sécurisée .

Puisque l'échange de messages passent dans le système **producteur/consommateur**, il faut sécuriser les accès de ces derniers grâce à la couche ssl configurée dans le serveur.

Cela consiste principalement à rajouter les **paramètres de sécurité du ssl** dans le code du Producer/Consumer et de spécifier que le protocole de sécurité utilisé dans les échanges est ssl.

La méthode java utilisée ici est **props.put** est utilisée pour ajouter une paire clé-valeur à un objet de type Properties en Java. Cette méthode prend deux arguments: la clé et la valeur correspondante.

```
public class ProducerCreator {  
    public static Producer<String, String> createProducer() {  
  
        Properties props = new Properties();  
        props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, IKafkaConst  
        props.put(ProducerConfig.CLIENT_ID_CONFIG, IKafkaConstants.CLI  
        props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, StringSe  
        props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, String  
        // Configurer la sécurité SSL  
        props.put("security.protocol", "SSL");  
        props.put("ssl.keystore.location", "/home/khady/sslDir/kafka.c  
        props.put("ssl.keystore.password", "pwdtpkafka");  
        props.put("ssl.key.password", "pwdtpkafka");  
        props.put("ssl.truststore.location", "/home/khady/sslDir/kafka  
        props.put("ssl.truststore.password", "pwdtpkafka");  
  
public class ConsumerCreator {  
    public static Consumer<String, String> createConsumer() {  
        props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, IKafkaConst  
        props.put(ConsumerConfig.GROUP_ID_CONFIG, IKafkaConstants.GROU  
        props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, String  
        props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG, Stri  
        props.put(ConsumerConfig.MAX_POLL_RECORDS_CONFIG, IKafkaConsta  
        props.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, "false");  
        props.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, IKafkaConst  
  
        // Configurer la sécurité SSL  
        props.put("security.protocol", "SSL");  
        props.put("ssl.keystore.location", "/home/khady/sslDir/kafka.c  
        props.put("ssl.keystore.password", "pwdtpkafka");  
        props.put("ssl.key.password", "pwdtpkafka");  
        props.put("ssl.truststore.location", "/home/khady/sslDir/kafka  
        props.put("ssl.truststore.password", "pwdtpkafka");
```

Après cela on va configurer le fichier **docker-compose.yml**. Cela garantit la sécurisation des communications entre les composants Kafka, assurant le chiffrement des données en transit et l'**authentification mutuelle** entre les parties. Elle facilite également la conformité aux normes de sécurité et simplifie le déploiement et la gestion de l'environnement Kafka sécurisé.

Voici une capture du code rajouté dans le fichier docker-compose.yml : Ce sont les paramètres de configuration du ssl.

```
KAFKA_JMX_PORT: 9101
KAFKA_SSL_KEYSTORE_LOCATION: /etc/kafka/secrets/kafka.server.key
KAFKA_SSL_KEYSTORE_PASSWORD: pwdtpkafka
KAFKA_SSL_KEY_PASSWORD: pwdtpkafka
KAFKA_SSL_TRUSTSTORE_LOCATION: /etc/kafka/secrets/kafka.server.t
KAFKA_SSL_TRUSTSTORE_PASSWORD: pwdtpkafka
KAFKA_SSL_CLIENT_AUTH: required
KAFKA_SECURITY_INTER_BROKER_PROTOCOL: SSL
volumes:
- /home/khady/sslDir:/etc/kafka/secrets
```

Puisque le protocole ssl fonctionne par défaut dans le port 9093, on change aussi l'adresse du serveur dans le fichier de constantes IKafkaConstants. On y stocke aussi les chemins vers les stores.

```
public interface IKafkaConstants {
    public static String KAFKA_BROKERS = "localhost:9093";
```

```
public static String SSL_KEYSTORE_LOCATION = "/home/khady/sslDir/k
public static String SSL_KEYSTORE_PASSWORD = "pwdtpkafka";
public static String SSL_KEY_PASSWORD = "pwdtpkafka";
public static String SSL_TRUSTSTORE_LOCATION = "/home/khady/sslDir
public static String SSL_TRUSTSTORE_PASSWORD = "pwdtpkafka";
```

Voilà maintenant on a complètement sécurisé notre serveur Apache Kafka avec le rajout d'une couche de sécurité ssl.

### III- [DEMO] : Test de notre serveur sécurisé

On va maintenant tester l'application

1. On lance le serveur zookeeper

```
(khady@kali)-[~/kafka/bin]$ ./zookeeper-server-start.sh ../config/zookeeper.properties
Picked up _JAVA_OPTIONS: -Dawt.useSystemAAFontSettings=on -Dswing.aatext=true
[2024-05-29 16:13:53,650] INFO Reading configuration from: ../config/zookeeper.properties (org.apache.zookeeper.server.quorum.QuorumPeerConfig)
[2024-05-29 16:13:53,656] WARN ../config/zookeeper.properties is relative. Prepend ./ to indicate that you're sure! (org.apache.zookeeper.server.quorum.QuorumPeerConfig)
[2024-05-29 16:13:53,660] INFO System.out.println("Record offset:" + record.offset)
[2024-05-29 16:13:53,664] INFO clientPortAddress is 0.0.0.0:2181 (org.apache.zookeeper.server.quorum.QuorumPeerConfig)
[2024-05-29 16:13:53,664] INFO secureClientPort is not set (org.apache.zookeeper.server.quorum.QuorumPeerConfig)
[2024-05-29 16:13:53,664] INFO observerMasterPort is not set (org.apache.zookeeper.server.quorum.QuorumPeerConfig)
[2024-05-29 16:13:53,664] INFO metricsProvider.className is org.apache.zookeeper.metrics.impl.DefaultMetricsProvider (org.apache.zookeeper.server.quorum.QuorumPeerConfig)
```

## 2. On lance le serveur de broker kafka

```
(khady@kali)-[~/kafka/bin]$ ./kafka-server-start.sh ../config/server.properties
Picked up _JAVA_OPTIONS: -Dawt.useSystemAAFontSettings=on -Dswing.aatext=true
[2024-05-29 16:14:57,268] INFO Registered kafka:type=kafka.Log4jController MBean (kafka.utils.Log4jControllerRegistration$)
[2024-05-29 16:14:58,064] INFO Setting -D jdk.tls.rejectClientInitiatedRenegotiation=true to disable client-initiated TLS renegotiation (org.apache.zookeeper.common.X509Util)
[2024-05-29 16:14:58,309] INFO Registered signal handlers for TERM, INT, HUP (org.apache.kafka.common.utils.LoggingSignalHandler)
[2024-05-29 16:14:58,313] INFO starting (kafka.server.KafkaServer)
[2024-05-29 16:14:58,320] INFO Connecting to zookeeper on localhost:2181 (kafka.server.KafkaServer)
```

- On voit dans la sortie les paramètres de sécurité du ssl

```
49     socket.send.buffer.bytes = 102400
50     ssl.allow.dn.changes = false
51     ssl.allow.san.changes = false
52     ssl.cipher.suites = []
53     ssl.client.auth = required
54     ssl.enabled.protocols = [TLSv1.2, TLSv1.3]
55     ssl.endpoint.identification.algorithm = https
56     ssl.engine.factory.class = null
57     ssl.key.password = [hidden]
58     ssl.keymanager.algorithm = SunX509
59     ssl.keystore.certificate.chain = null
60     ssl.keystore.key = null
61     ssl.keystore.location = /home/khady/sslDir/kafka.server.keystore.jks
62     ssl.keystore.password = [hidden]
63     ssl.keystore.type = JKS
64     ssl.principal.mapping.rules = DEFAULT
65     ssl.protocol = TLSv1.3
66     ssl.provider = null
67     ssl.secure.random.implementation = null
68     ssl.trustmanager.algorithm = PKIX
```

- On crée un topic : lorsque SSL est activé, le serveur Kafka peut nécessiter une configuration supplémentaire pour les autorisations d'accès aux topics. Cela peut impliquer de définir des politiques de contrôle d'accès basées sur les certificats SSL pour s'assurer que seules les entités authentifiées peuvent créer ou accéder à certains topics. En créant un topic donc on lui passe maintenant un fichier config-ssl.properties où on a

stocké les paramètres de sécurité du ssl. Le fichier est enregistré dans le même répertoire que server.properties c'est à dire kafka/config



```
config-ssl.properties
~[kafka/config]

security.protocol=SSL
ssl.truststore.location=/home/khady/sslDir/kafka.client.truststore.jks
ssl.truststore.password=pwdtpkafka
ssl.keystore.location=/home/khady/sslDir/kafka.client.keystore.jks
ssl.keystore.password=pwdtpkafka
ssl.key.password=pwdtpkafka
```

- On peut maintenant créer le topic, on spécifie bien l'adresse localhost:9093 où s'exécute le ssl

```
(khady㉿kali)-[~/kafka/bin]
$ ./kafka-topics.sh --bootstrap-server localhost:9093 --create --replication-factor 1 --partitions 1 --topic HT-topic4 --command-config /home/khady/kafka/config/config-ssl.properties
Picked up _JAVA_OPTIONS: -Dawt.useSystemAAFontSettings=on -Dswing.aatext=true
Created topic HT-topic4.
```

- On modifie le nom du topic dans le fichier des constantes IKafkaContents

```
public static String TOPIC_NAME="HT-topic4";
```

3. On compile et lance la classe App.java qui instancie un producteur, un consommateur, l'envoi et la réception de messages

On a utilisé mvn pour compiler et lancer le fichier java

- La compilation

```
(khady㉿kali)-[~/TPKafka/kafka-producer-consumer/kafka-producer-consumer-example]
$ mvn clean compile
Picked up _JAVA_OPTIONS: -Dawt.useSystemAAFontSettings=on -Dswing.aatext=true
[INFO] Scanning for projects...
[INFO] 
[INFO] --- < com.uphf.kafka:kafka-producer-consumer-exemple > -----
[INFO] Building kafka-producer-consumer-exemple 0.0.1
[INFO] --- [ jar ] ---
[INFO] 
[INFO] --- maven-clean-plugin:2.5:clean (default-clean) @ kafka-producer-consumer-exemple ---
[INFO] Deleting /home/khady/TPKafka/kafka-producer-consumer/kafka-producer-consumer-exemple/target
[INFO] 
[INFO] --- maven-resources-plugin:2.6:resources (default-resources) @ kafka-producer-consumer-exemple ---
```

```
[INFO] 
[INFO] BUILD SUCCESS
[INFO] 
[INFO] Total time:  3.963 s
[INFO] Finished at: 2024-05-29T16:25:05-05:00
[INFO] 
```

- L'exécution

```

[~(khady@kali)-[~/TPKafka/kafka-producer-consumer/kafka-producer-consumer-example]
└$ mvn exec:java -Dexec.mainClass="com.uphf.kafka.App"
Picked up _JAVA_OPTIONS: -Dawt.useSystemAAFontSettings=on -Dswing.aatext=true
[INFO] Scanning for projects...
[INFO] [INFO] -----< com.uphf.kafka:kafka-producer-consumer-exemple >-----
[INFO] Building kafka-producer-consumer-exemple 0.0.1
[INFO] -----[ jar ]-----
[INFO] --- exec-maven-plugin:3.2.0:java (default-cli) @ kafka-producer-consumer-exemple ---
LANCEMENT DU PROCUCER
[com.uphf.kafka.App.main()] INFO org.apache.kafka.clients.producer.ProducerConfig - ProducerConfig values:

```

#### - Sortie ssl

```

ssl.cipher.suites = null
ssl.enabled.protocols = [TLSv1.2, TLSv1.3]
ssl.endpoint.identification.algorithm = https value() + record.value()
ssl.engine.factory.class = null
ssl.key.password = [hidden] println("Record partition " + record.partition)
ssl.keymanager.algorithm = SunX509
ssl.keystore.certificate.chain = null("Record offset " + record.offset)
ssl.keystore.key = null
ssl.keystore.location = /home/khady/sslDir/kafka.client.keystore.jks
ssl.keystore.password = [hidden]
ssl.keystore.type = JKS
ssl.protocol = TLSv1.3
ssl.provider = null
ssl.secure.random.implementation = null
ssl.trustmanager.algorithm = PKIX
ssl.truststore.certificates = null
ssl.truststore.location = /home/khady/sslDir/kafka.client.truststore.jks
ssl.truststore.password = [hidden]
ssl.truststore.type = JKS

```

#### - Envoi des messages par le producteur

```

khady@kali: ... ✘ khady@kali: ... ✘ khady@kali: ... ✘ khady@kali: ... ✘ khady@kali: ... ✘
ssl.keystore.password = [hidden]
ssl.keystore.type = JKS
ssl.protocol = TLSv1.3
ssl.provider = null
ssl.secure.random.implementation = null
ssl.trustmanager.algorithm = PKIX
ssl.truststore.certificates = null
ssl.truststore.location = /home/khady/sslDir/kafka.client.truststore.jks
ssl.truststore.password = [hidden]
ssl.truststore.type = JKS
transaction.timeout.ms = 60000
value.serializer = class org.apache.kafka.common.serialization.StringSerializer
com.uphf.kafka.App.main() INFO org.apache.kafka.common.telemetry.internals.KafkaMetricsCollector - initializing Kafka metrics collector
com.uphf.kafka.App.main() INFO org.apache.kafka.clients.producer.KafkaProducer - [Producer clientId=client1] Instantiated an idempotent producer.
com.uphf.kafka.App.main() INFO org.apache.kafka.common.utils.AppInfoParser - Kafka version: 3.7.0
com.uphf.kafka.App.main() INFO org.apache.kafka.common.utils.AppInfoParser - Kafka commitId: 2ae524ed625438c5
com.uphf.kafka.App.main() INFO org.apache.kafka.common.utils.AppInfoParser - Kafka startTimeMs: 1717024705136
kafka-producer-network-thread | client1 INFO org.apache.kafka.clients.Metadata - [Producer clientId=client1] Cluster ID: BNmSVQqRwCsRpg7FfvMg
kafka-producer-network-thread | client1 INFO org.apache.kafka.clients.producer.internals.TransactionManager - [Producer clientId=client1] ProducerId set to 1002 with epoch 0
enregistrement envoyer avec clé 0 vers la partition 0 Et l'offset 3
enregistrement envoyer avec clé 1 vers la partition 0 Et l'offset 4
enregistrement envoyer avec clé 2 vers la partition 0 Et l'offset 5
enregistrement envoyer avec clé 3 vers la partition 0 Et l'offset 6
enregistrement envoyer avec clé 4 vers la partition 0 Et l'offset 7
enregistrement envoyer avec clé 5 vers la partition 0 Et l'offset 8
enregistrement envoyer avec clé 6 vers la partition 0 Et l'offset 9
enregistrement envoyer avec clé 7 vers la partition 0 Et l'offset 10
enregistrement envoyer avec clé 8 vers la partition 0 Et l'offset 11
enregistrement envoyer avec clé 9 vers la partition 0 Et l'offset 12

```

#### - Consommation des messages par le consommateur

```
[com.uphf.kafka.App.main()] INFO org.apache.kafka.clients.consumer.internals.ConsumerCoordinator - [Consumer clientId=consumer-consumerGroup10-1, groupId=consumerGroup10] Notifying assignor about the new Assignment(partitions=[HT-topic4-0])
[com.uphf.kafka.App.main()] INFO org.apache.kafka.clients.consumer.internals.ConsumerRebalanceListenerInvoker - [Consumer clientId=consumer-consumerGroup10-1, groupId=consumerGroup10] Adding newly assigned partitions: HT-topic4-0
[com.uphf.kafka.App.main()] INFO org.apache.kafka.clients.consumer.internals.ConsumerCoordinator - [Consumer clientId=consumer-consumerGroup10-1, groupId=consumerGroup10] Found no committed offset for partition HT-topic4-0
[com.uphf.kafka.App.main()] INFO org.apache.kafka.clients.consumer.internals.SubscriptionState - [Consumer clientId=consumer-consumerGroup10-1, groupId=consumerGroup10] Resetting offset for partition HT-topic4-0 to position FetchPosition{offset=0, epochEpoch=Optional.empty, currentLeader=leaderAndEpoch[leader(optional=localhost:9093 {id: 0 rack: null}), epoch=0]}.
Record Key ID=0
Record value Enregistrement N° 0
Record partition 0
Record offset 0
Record Key ID=3
Record value Enregistrement N° 3
Record partition 0
Record offset 1
Record Key ID=6
Record value Enregistrement N° 6
Record partition 0
Record offset 2
Record Key ID=0
Record value Enregistrement N° 0
Record partition 0
Record offset 3
Record Key ID=1
Record value Enregistrement N° 1
Record partition 0
Record offset 4
Record Key ID=2
Record value Enregistrement N° 2
Record partition 0
Record offset 5
Record Key ID=3
Record value Enregistrement N° 3
Record partition 0
Record offset 6
```

## Réception de tous les messages

```
Record value Enregistrement N° 93
Record partition 0
Record offset 96
Record Key ID=94    public static String INTERFACE
Record value Enregistrement N° 94
Record partition 0
Record offset 97
Record Key ID=95    public static Integer MESSAGE
Record value Enregistrement N° 95
Record partition 0
Record offset 98    public static String CLIENT
Record Key ID=96    public static String TOPIC
Record value Enregistrement N° 96
Record partition 0
Record offset 99    public static String GROUP
Record Key ID=97    public static Integer MAX_OFFSET
Record partition 0
Record offset 100
Record Key ID=98    public static Integer MAX_OFFSET
Record value Enregistrement N° 98
Record partition 0
Record offset 101
Record Key ID=99    public static String OFFSET
Record value Enregistrement N° 99
Record partition 0
Record offset 102    public static Integer MAX_OFFSET
```

Parfait, notre configuration a bien marché et la communication au sein du serveur Apache est sécurisée.

## Iv- Développement d'un consommateur supplémentaire client et utilisation des partitions

Pour cette partie du projet, on a développé un consommateur supplémentaire capable de consommer les messages d'une partition spécifique et transformer le producteur pour qu'il envoie des messages équitablement sur toutes les partitions du topic. Pour des soucis de clarté la classe principale pour exécuter le projet incluant le consommateur supplémentaire est `AppWithMoreConsumers.java`

### Création d'un consommateur supplémentaire:

1. **Configuration du consommateur** : On commence par créer un nouveau consommateur Kafka dans notre projet Java. On configure ce consommateur pour qu'il se connecte à notre broker sécurisé via SSL, en incluant les fichiers de certificats nécessaires.
2. **Implémentation du consommateur** : On implémente le consommateur de sorte qu'il puisse accepter un numéro de partition en paramètre. Cela permet de spécifier quelle partition on souhaite consommer. Le consommateur doit être capable de consommer les messages d'une partition donnée.

### Transformation du producteur:

1. **Configuration du producteur** : Comme pour le consommateur, on configure le producteur pour qu'il se connecte au broker sécurisé via SSL. Les configurations SSL incluent les chemins vers les **fichiers keystore** et **truststore**, ainsi que les **mots de passe** correspondants.
2. **Modification de l'envoi des messages** : On modifie le producteur pour qu'il envoie des messages sur différentes partitions de manière équitable en utilisant la formule **index % PartitionCount**. Cela garantit que les messages sont répartis uniformément sur l'ensemble des partitions du topic. L'**index** représente le numéro de message et il boucle pour assurer une répartition équilibrée.

## Ajustement du broker

1. **Création du topic avec plusieurs partitions** : Lors de la création du topic, on spécifie le nombre de partitions souhaité (supérieur à 1), en utilisant les commandes Kafka appropriées.

```
public static String TOPIC_NAME="HT-topic5";
```

```
[(khady@kali)-[~/kafka/bin]] integer MAX.NO MESSAGE FOUND COUNT=166
$ ./kafka-topics.sh --bootstrap-server localhost:9093 --create --replication-factor 1 --partitions 3 --topic HT-topic5 --command-config /home/khady/kafka/config/config-ssl.properties
Picked up _JAVA_OPTIONS: -Dawt.useSystemAAFontSettings=on -Dswing.aatext=true
Created topic HT-topic5.      integer MAX.PART.REC=1,
```

## Test et Exécution:

```
[(khady@kali)-[~/TPKafka/kafka-producer-consumer/kafka-producer-consumer-exemple]]
$ mvn exec:java -Dexec.mainClass="com.uphf.kafka.App2" -Dexec.args="partitionConsumer 2"
Picked up _JAVA_OPTIONS: -Dawt.useSystemAAFontSettings=on -Dswing.aatext=true
[INFO] Scanning for projects...
[INFO] ------------------------------------------------------------------------
[INFO] --- exec-maven-plugin:3.2.0:java (default-cli) @ kafka-producer-consumer-exemple ---
[INFO] Building kafka-producer-consumer-exemple 0.0.1
[INFO] ------------------------------------------------------------------------
[INFO] --- [ jar ] ---
[INFO] --- exec-maven-plugin:3.2.0:java (default-cli) @ kafka-producer-consumer-exemple ---
LANCEMENT DU PRODUCER
[com.uphf.kafka.App2.main()] INFO org.apache.kafka.clients.producer.ProducerConfig - ProducerConfig values:
    acks = -1
    auto.include.jmx.reporter = true
    batch.size = 16384
    bootstrap.servers = [localhost:9093]
    buffer.memory = 33554432
    client.dns.lookup = use_all_dns_ips
    client.id = client1
```

```
Record Key ID=0
Record value Enregistrement N° 0
Record partition 0
Record offset 0
Record Key ID=3
Record value Enregistrement N° 3
Record partition 0
Record offset 1
Record Key ID=6
Record value Enregistrement N° 6
Record partition 0
Record offset 2
Record Key ID=9
Record value Enregistrement N° 9
Record partition 0
```

```
Record value Enregistrement N° 2
Record partition 2
Record offset 0
Record Key ID=5
Record value Enregistrement N° 5
Record partition 2
Record offset 1
Record Key ID=8
Record value Enregistrement N° 8
Record partition 2
Record offset 2
Record Key ID=11
Record value Enregistrement N° 11
Record partition 2
Record offset 3
Record Key ID=14
```

En suivant ces étapes, nous aurons implémenté un consommateur supplémentaire capable de lire les messages d'une partition spécifique. Le broker Kafka est également configuré pour gérer plusieurs partitions, assurant une répartition équitable des messages et une consommation efficace.

## v- 2ème Version du Projet:

Pour un 2ème cas, on a travaillé sur une version plus indépendante des ressources données par notre professeur et on s'est débrouillé pour obtenir les résultats attendus.

- **Pré-requis**
  - Java installé sur la machine.
  - Apache Kafka téléchargé et installé.
  - OpenSSL installé pour générer des certificats.
  - Configuration de Kafka pour supporter SSL.
- **Initiation de toutes les classes utilisées:**

```
import java.util.Properties;
import java.util.concurrent.ExecutionException;

public class SSLKafkaProducer {
    Run | Debug
    public static void main(String[] args) {
        Properties props = new Properties();
        props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, StringSerializer.class);
        props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, StringSerializer.class);

        // SSL configuration
        props.put("security.protocol", "SSL");
        props.put("ssl.truststore.location", "C:/TPKAFKA/server_certs/cacerts.jks");
        props.put("ssl.truststore.password", "raniabadi");
        props.put("ssl.keystore.location", "C:/TPKAFKA/server_certs/producer.jks");
        props.put("ssl.keystore.password", "raniabadi");
```

- **Configuration des propriétés :**
  - Un objet **Properties** est créé pour stocker les configurations nécessaires pour le producteur Kafka.
  - Les configurations de base incluent l'adresse du serveur Kafka (**localhost:9092**) et les sérialiseurs de clés et de valeurs en tant que chaînes de caractères.
  - Les configurations SSL sont ajoutées pour sécuriser les communications :

- `security.protocol` est défini sur `SSL`.
- Les emplacements des fichiers `truststore` et `keystore` ainsi que leurs mots de passe respectifs sont spécifiés.

- **Création du producteur Kafka :**

- Une instance de `KafkaProducer` est créée en utilisant les propriétés configurées.

- **Vérification de l'existence du topic :**

- Un `AdminClient` est créé pour vérifier si le topic `firsttopic` existe.
- La méthode `describeTopics` est utilisée pour obtenir des informations sur le topic.
- Si le topic existe, un message est affiché indiquant que les messages seront envoyés.

- **Envoi des messages :**

- Une boucle `for` est utilisée pour envoyer 10 messages au topic `firsttopic`.
- Pour chaque message, un `ProducerRecord` est créé avec une clé et une valeur.
- Le message est envoyé de manière asynchrone avec un rappel (`Callback`) pour gérer les résultats :
  - Si une exception se produit, un message d'erreur est affiché.
  - Sinon, les détails de l'enregistrement envoyé (clé, partition, offset) sont affichés.

- **Fermeture des ressources :**

- Enfin, les clients `AdminClient` et `KafkaProducer` sont fermés pour libérer les ressources.

```
public class SSLKafkaConsumer {
    Run | Debug
    public static void main(String[] args) {
        Properties props = new Properties();
        props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        props.put(ConsumerConfig.GROUP_ID_CONFIG, "test-group");
        props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, StringDeserializer.class);
        props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG, StringDeserializer.class);
        props.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest");

        // SSL configuration
        props.put("security.protocol", "SSL");
        props.put("ssl.truststore.location", "C:/TPKAFKA/server_certs/client_truststore.jks");
        props.put("ssl.truststore.password", "raniabadi");
        props.put("ssl.keystore.location", "C:/TPKAFKA/server_certs/client_keystore.jks");
        props.put("ssl.keystore.password", "raniabadi");
        props.put("ssl.key.password", "raniabadi");
    }
}
```

Ce code est un consommateur Kafka qui utilise SSL pour sécuriser la communication avec le cluster Kafka:

- **Configuration des propriétés :**

- Un objet **Properties** est créé pour stocker les configurations nécessaires pour le consommateur Kafka.
- Les configurations de base incluent :
  - L'adresse du serveur Kafka (**localhost:9092**).
  - Le groupe de consommateurs (**test-group**).
  - Les déserialiseurs de clés et de valeurs en tant que chaînes de caractères.
  - Le reset d'offset automatique (**earliest**) pour lire les messages depuis le début si aucun offset n'est initialisé.

- **Configuration SSL :**

- Les propriétés SSL sont ajoutées pour sécuriser les communications :
  - **security.protocol** est défini sur **SSL**.
  - Les emplacements des fichiers **truststore** et **keystore** ainsi que leurs mots de passe respectifs sont spécifiés.

- **Création du consommateur Kafka :**

- Une instance de **KafkaConsumer** est créée en utilisant les propriétés configurées.

- **Attribution des partitions :**

- Trois partitions spécifiques du sujet `firsttopic` sont assignées au consommateur :
  - `partition0` (partition 0 de `firsttopic`)
  - `partition1` (partition 1 de `firsttopic`)
  - `partition2` (partition 2 de `firsttopic`)

- **Boucle de consommation :**

- Une boucle infinie est utilisée pour poller les messages des partitions assignées.
- La méthode `poll` avec un timeout de 1000 millisecondes est utilisée pour récupérer les enregistrements.
- Pour chaque enregistrement récupéré, les informations de clé, valeur, partition et offset sont affichées.

```
public class PartitionKafkaConsumer {  
    Run | Debug  
    public static void main(String[] args) {  
        if (args.length != 1) {  
            System.out.println("Please provide the partition number");  
            System.exit(1);  
        }  
  
        int partitionNumber = Integer.parseInt(args[0]);  
  
        Properties props = new Properties();
```

Ce code est un consommateur Kafka qui consomme des messages d'une partition spécifique d'un sujet Kafka.

- **Validation des arguments :**

- Le programme vérifie que le nombre d'arguments est correct (1 argument attendu pour le numéro de partition).
- Si le nombre d'arguments est incorrect, un message est affiché et le programme se termine.

- **Lecture du numéro de partition :**

- Le numéro de partition est lu à partir des arguments de la ligne de commande et converti en entier.

- **Configuration des propriétés :**

- Un objet `Properties` est créé pour stocker les configurations nécessaires pour le consommateur Kafka.
- Les configurations de base incluent :
  - L'adresse du serveur Kafka (`localhost:9092`).
  - Les déserialiseurs de clé et de valeur en tant que chaînes de caractères.
  - L'ID du groupe de consommateurs (`test-group`).
  - La stratégie de réinitialisation de l'offset (`earliest`), ce qui signifie que le consommateur commencera à lire à partir du début du journal si aucun offset précédent n'est trouvé.
- Les configurations SSL sont ajoutées pour sécuriser les communications :
  - `security.protocol` est défini sur `SSL`.
  - Les emplacements des fichiers `truststore` et `keystore` ainsi que leurs mots de passe respectifs sont spécifiés.

- **Création du consommateur Kafka :**

- Une instance de `KafkaConsumer` est créée en utilisant les propriétés configurées.

- **Assignation de la partition :**

- Un objet `TopicPartition` est créé en utilisant le sujet `firstrtopic` et le numéro de partition spécifié.
- Le consommateur est configuré pour consommer uniquement la partition spécifiée en utilisant la méthode `assign`.

- **Boucle de consommation :**

- Une boucle infinie est utilisée pour poller les messages de la partition spécifiée.
- Les messages sont récupérés avec une temporisation de 1000 millisecondes.
- Pour chaque message consommé, les détails du message (clé, valeur, partition, offset) sont affichés.

```

public class PartitionKafkaProducer {
    Run | Debug
    public static void main(String[] args) {
        Properties props = new Properties();
        props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, StringSerializer.class.getName());
        props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, StringSerializer.class.getName());

        // SSL configuration
        props.put("security.protocol", "SSL");
        props.put("ssl.truststore.location", "C:/TPKAFKA/server_certs/client.truststore");
        props.put("ssl.truststore.password", "raniabadi");
        props.put("ssl.keystore.location", "C:/TPKAFKA/server_certs/client.keystore");
        props.put("ssl.keystore.password", "raniabadi");
        props.put("ssl.key.password", "raniabadi");
    }
}

```

Ce code est un producteur Kafka qui envoie des messages à différentes partitions d'un sujet Kafka de manière cyclique

- **Configuration des propriétés :**

- Un objet `Properties` est créé pour stocker les configurations nécessaires pour le producteur Kafka.
- Les configurations de base incluent l'adresse du serveur Kafka (`localhost:9092`), et les sérialiseurs de clé et de valeur en tant que chaînes de caractères.
- Les configurations SSL sont ajoutées pour sécuriser les communications :
  - `security.protocol` est défini sur `SSL`.
  - Les emplacements des fichiers `truststore` et `keystore` ainsi que leurs mots de passe respectifs sont spécifiés.

- **Création du producteur Kafka :**

- Une instance de `KafkaProducer` est créée en utilisant les propriétés configurées.

- **Gestion des partitions :**

- Un `AtomicInteger` nommé `partitionCounter` est utilisé pour compter les partitions de manière cyclique.
- Le nombre de partitions (`numPartitions`) est défini à 3 (vous pouvez le changer selon le nombre de partitions de votre sujet).

- **Envoi des messages :**
- Une boucle `for` est utilisée pour envoyer 10 messages au sujet `firsttopic`.
- Pour chaque message, la partition est déterminée en utilisant le `partitionCounter` pour distribuer les messages de manière cyclique entre les partitions.
- Un `ProducerRecord` est créé avec une clé et une valeur, ainsi qu'avec la partition déterminée.
- Le message est envoyé de manière asynchrone avec un rappel (`Callback`) pour gérer les résultats :
  - Si une exception se produit, un message d'erreur est affiché.
  - Sinon, les détails de l'enregistrement envoyé (topic, partition, offset) sont affichés.

## 1. La sécurisation de la partie serveur & client:

Nous avons suivi les mêmes étapes établies dans la 1ère version pour obtenir une connexion sécurisée via SSL/TLS:

```
C:\TPKAFKA>docker logs tpkafka-kafka-1
==> User
uid=1000(appuser) gid=1000(appuser) groups=1000(appuser)
==> Configuring ...
SSL is enabled.
```

```

PS C:\TPKAFKA> openssl s_client -connect localhost:9092
Connecting to ::1
CONNECTED(000001E4)
Can't use SSL_get_servername
depth=0 CN=localhost
verify error:num=18:self-signed certificate
verify return:1
depth=0 CN=localhost
verify return:1
---
Certificate chain
 0 s:CN=localhost
    i:CN=localhost
      a:PKEY: rsaEncryption, 2048 (bit); sigalg: RSA-SHA256
      v:NotBefore: May 27 23:52:42 2024 GMT; NotAfter: May 27 23:52:42 2025 GMT
---
Server certificate
-----BEGIN CERTIFICATE-----
MIICxzCCAa+gAwIBAgIEXI1/jDANBgkqhkiG9w0BAQsFADAUMRIwEAYDVQQDEwls
b2NhbGhv3QwHhcNMjQwNTI3MjM1MjQyWhcNMjUwNTI3MjM1MjQyWjAUMRIwEAYD
VQQDEwlsb2NhbGhv3QwggEiMA0GCSqGSIb3DQEBAQUAA4IBDwAwggEKAoIBAQCl
VbX+FQiad86TgdcRCn4zu0q2/JT7Q/Io0yf9t1FTmZ8AmJo8B3CN6k1908I/j9y6
1XOFVxnPknGhqOnN6JsqfZPhr3o+f4g/tXVM4nd0yMYYwtpz+s+GZBLkLb95ki
Q6CNWG7HSzvcnCp2aJIByh9upkPiZFVE8mgSCeSQuAqDaaI2za0Eod9CtkKRwI1U
DbHVA4QZvPQZuQ5NsTNDgcjZWiAqBQHsMOLBk+vzfxvCdPXyHo4Qj0u/AX6wACV
cfSFMcCtpCKX05RrqyNCytYrcxtfUHoezdWqoaE1Zu5V6BktTFwWfvv60Zj+j
J0j7oKC8Xw47GEREnt//AgMBAAGjITAfMB0GA1UdDgQWBQ4qbSyySiI RdDz7kxj
t09Q47+JDjANBgkqhkiG9w0BAQsFAAOCAQEAiQh9fCTXRsjGC8488/OjuRX1oMd
QYfIMZ10fUSGE/V2rtKD9eg5paL7JvxNra721bwMzQoeuU09vE5I1kn/1p56wm6
KQhmMfxnHnNzcGjy1YHofZuUnP9zTHabaTXn8NmRYFJdbdFcuzJAKJqMUFBpUh4
Ioc2jt1p6h0rJRTNv9yUsPs05vUndEENixlJMaRpnoJA8b14zprQJEIRI6nVTCz
itGYOz2nI7WB9s4Lrlsbelsrv0tmd0hNbiFij7PwI2QexvLxDswL+4A6XjzAx
-----END CERTIFICATE-----

```

#### - Envoi des messages par le producteur:

```
[main] INFO orgjava -cp ".;C:\TPKAFKA\libs\kafka-clients-2.7.0.jar;C:\TPKAFKA\libs\slf4j-api-1.7.30.jar;C:\TPKAFKA\li
bs\slf4j-simple-1.7.30.jar" SSLKafkaProducer
```

```

socket.connection.setup.timeout.ms = 10000
ssl.cipher.suites = null
ssl.enabled.protocols = [TLSv1.2, TLSv1.3]
ssl.endpoint.identification.algorithm = https
ssl.engine.factory.class = null
ssl.key.password = [hidden]
ssl.keymanager.algorithm = SunX509
ssl.keystore.certificate.chain = null
ssl.keystore.key = null
ssl.keystore.location = C:/TPKAFKA/server_certs/client.keystore.jks
ssl.keystore.password = [hidden]
ssl.keystore.type = JKS
ssl.protocol = TLSv1.3
ssl.provider = null
ssl.secure.random.implementation = null
ssl.trustmanager.algorithm = PKIX

```

```
er with timeoutMillis = 9223372036854775807 ms.  
Enregistrement envoyé avec clé 0 vers la partition 0 et l'offset 66  
Enregistrement envoyé avec clé 1 vers la partition 0 et l'offset 67  
Enregistrement envoyé avec clé 2 vers la partition 0 et l'offset 68  
Enregistrement envoyé avec clé 3 vers la partition 0 et l'offset 69  
Enregistrement envoyé avec clé 4 vers la partition 0 et l'offset 70  
Enregistrement envoyé avec clé 5 vers la partition 0 et l'offset 71  
Enregistrement envoyé avec clé 6 vers la partition 0 et l'offset 72  
Enregistrement envoyé avec clé 7 vers la partition 0 et l'offset 73  
Enregistrement envoyé avec clé 8 vers la partition 0 et l'offset 74  
Enregistrement envoyé avec clé 9 vers la partition 0 et l'offset 75  
[main] INFO org.apache.kafka.common.metrics.Metrics - Metrics scheduler closed
```

- *Consommation des messages par le consommateur :*

```
PS C:\TPKAFKA> java -cp "C:\TPKAFKA\libs\kafka-clients-2.7.0.jar;C:\TPKAFKA\libs\slf4j-api-1.7.30.jar;C:\TPKAFKA\libs\slf4j-simple-1.7.30.jar;." SSLKafkaConsumer
```

```
Record key ID=0  
Record value Enregistrement N?° 0  
Record partition 0  
Record offset 80  
Record Key ID=1  
Record value Enregistrement N?° 1  
Record partition 0  
Record offset 81  
Record Key ID=2  
Record value Enregistrement N?° 2  
Record partition 0  
Record offset 82  
Record Key ID=3  
Record value Enregistrement N?° 3  
Record partition 0
```

## 2. Crédit d'un consommateur supplémentaire:

- La création d'un consommateur supplémentaire dans un groupe de consommateurs existant entraîne une répartition des partitions entre les consommateurs, permettant une consommation parallèle et équilibrée des messages. Les logs fourniront des informations détaillées sur les messages consommés, y compris les partitions et les offsets.
- Les partitions permettent à plusieurs consommateurs de lire des messages en parallèle. Dans un groupe de consommateurs, chaque consommateur peut lire une partition ou un sous-ensemble de partitions, ce qui augmente le débit global du traitement des messages.
- Par exemple, avec 3 partitions et 3 consommateurs, chaque consommateur peut lire indépendamment une partition, accélérant ainsi le traitement des messages.

- **Côté Producteur:**

```
PS C:\TPKAFKA> java -cp "C:\TPKAFKA\libs\kafka-clients-2.7.0.jar;C:\TPKAFKA\libs\slf4j-api-1.7.30.jar;C:\TPKAFKA\libs\slf4j-simple-1.7.30.jar;." PartitionKafkaProducer
```

```
ssl.enabled.protocols = [TLSv1.2, TLSv1.3]
ssl.endpoint.identification.algorithm = https
ssl.engine.factory.class = null
ssl.key.password = [hidden]
ssl.keymanager.algorithm = SunX509
ssl.keystore.certificate.chain = null
ssl.keystore.key = null
ssl.keystore.location = C:/TPKAFKA/server_certs/client.keystore.jks
ssl.keystore.password = [hidden]
ssl.keystore.type = JKS
ssl.protocol = TLSv1.3
ssl.provider = null
ssl.secure.random.implementation = null
ssl.trustmanager.algorithm = PKIX
ssl.truststore.certificates = null
ssl.truststore.location = C:/TPKAFKA/server_certs/client.truststore.jks
ssl.truststore.password = [hidden]
ssl.truststore.type = JKS
transaction.timeout.ms = 60000
transactional.id = null
```

```
Message sent to topic: firsttopic partition: 0 offset: 232
Message sent to topic: firsttopic partition: 0 offset: 233
Message sent to topic: firsttopic partition: 0 offset: 234
Message sent to topic: firsttopic partition: 0 offset: 235
Message sent to topic: firsttopic partition: 2 offset: 65
Message sent to topic: firsttopic partition: 2 offset: 66
Message sent to topic: firsttopic partition: 2 offset: 67
Message sent to topic: firsttopic partition: 1 offset: 43
Message sent to topic: firsttopic partition: 1 offset: 44
Message sent to topic: firsttopic partition: 1 offset: 45
```

- Les messages sont envoyés de manière cyclique ou en fonction de la clé de partitionnement spécifiée. Ici, les messages sont distribués sur 3 partitions (0, 1, et 2), ce qui montre comment Kafka répartit les messages pour équilibrer la charge.

- **Côté Consommateur:**

```
PS C:\TPKAFKA> java -cp "C:\TPKAFKA\libs\kafka-clients-2.7.0.jar;C:\TPKAFKA\libs\slf4j-api-1.7.30.jar;C:\TPKAFKA\libs\slf4j-simple-1.7.30.jar;." PartitionKafkaConsumer 1
setEpoch=Optional[0], currentLeader=LeaderAndEpoch{leader=Optional[localhost:9092 (id:1 partition:0)], epoch=1}
Consumed message: key = 1, value = Message 1, partition = 1, offset = 46
Consumed message: key = 4, value = Message 4, partition = 1, offset = 47
Consumed message: key = 7, value = Message 7, partition = 1, offset = 48
[main] ERROR org.apache.kafka.clients.consumer.internals.ConsumerCoordinator - [Consumer clientId=PartitionKafkaConsumer-1, group=group-1] Offset for partition 1-0 could not be updated: offset 46 is less than current offset 47 for the partition
```

- L'argument **1** à côté de **PartitionKafkaConsumer** dans la commande que exécutée spécifie le numéro de la partition que le consommateur doit lire.
- Le consommateur est configuré pour lire les messages de plusieurs partitions. Chaque ligne montre un message consommé avec des détails sur la partition et l'offset. La présence de plusieurs partitions permet au consommateur de lire les messages en parallèle, améliorant ainsi l'efficacité du traitement.

D'après analyse des exécutions , les multiples valeurs de partitions dans les résultats montrent la répartition et la parallélisation des messages et du traitement des messages dans Kafka, ce qui est essentiel pour la scalabilité, la tolérance aux pannes et l'efficacité du traitement.

## CONCLUSION

Ce projet nous a permis de comprendre et d'appliquer la sécurisation d'un cluster Kafka via SSL, assurant ainsi la confidentialité, l'authentification, et l'intégrité des données transmises. La démonstration pratique a validé l'efficacité de notre configuration, et le développement supplémentaire a montré comment gérer efficacement les partitions et les consommateurs multiples.

## ANNEXES

<https://medium.com/jinternals/kafka-ssl-setup-with-self-signed-certificate-part-1-c2679a57e16c>

<https://kafka.apache.org/documentation/#security>