eurotech
UNLEASHING THE POWER OF THE EDGE

Apache Kafka® Cloud Connector Documentation

Table of Contents

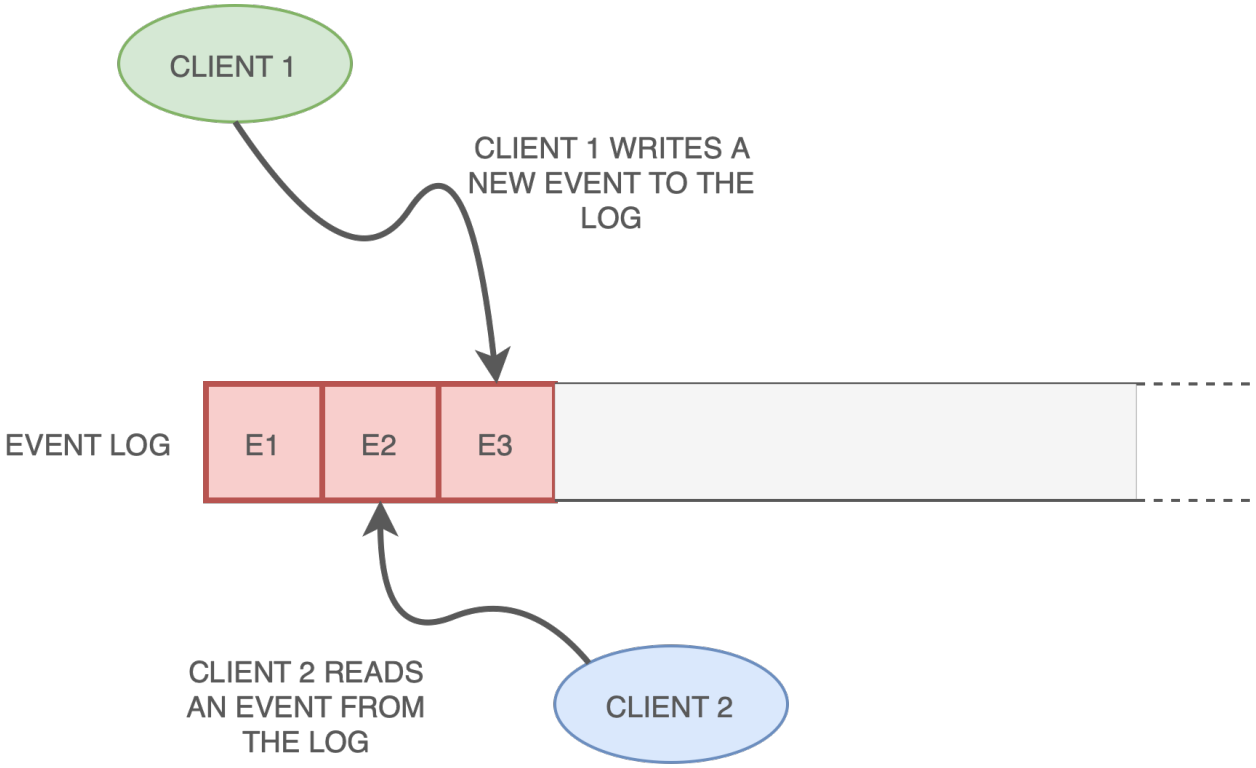
References

Home

Introduction

Version: 1.0.0

Apache Kafka® is an open-source framework that allows the reading, storing, and processing of data streams. It is designed to be deployed on distributed systems to allow high availability and scalability. The data exchanged in Kafka are **events**. Kafka Clients can read, process, and store events on a common distributed **event log**, which is kept aligned and synchronized by the Kafka Broker itself.



The Apache Kafka® Cloud Connector is a Kafka Client that allows interacting with a Kafka Broker by producing or consuming event streams via a **publisher/subscriber model**. The event stream is maintained in the broker and clients can publish, subscribe to, and process events. A good introduction to the fundamental Kafka concepts can be found at the Kafka Introduction *(https://kafka.apache.org/intro)*.

Prerequisites

- The Apache Kafka® Cloud Connector installed on the system.
- A running Kafka Broker with well-known topics.

- If running ESF < 7.2.0, an `org.eclipse.kura.cloud.CloudService` needs to be instantiated in order to use *Kura Protobuf* payload encoding (see Endpoint Configuration).

General concepts

The provided cloud endpoint encapsulates a `KafkaConsumer` *(https://kafka.apache.org/documentation/#consumerapi)* and a `KafkaProducer` *(https://kafka.apache.org/documentation/#producerapi)*.

Producers are those client applications that publish (write) events to Kafka, and **Consumers** are those that subscribe to (read and process) these events. In Kafka, producers and consumers are fully decoupled and agnostic of each other, which is a key design element to achieving the high scalability that Kafka is known for.

The Cloud Connection encapsulates an instance of a `KafkaConsumer` and an instance of a `KafkaProducer`. In order to have multiple producer or consumer instances you need to create multiple cloud connections. `CloudPublisher` or `CloudSubscriber` instances created using this Cloud Connector use the `KafkaProducer` or `KafkaConsumer` of the created Cloud Connection instance.

Kafka Topics

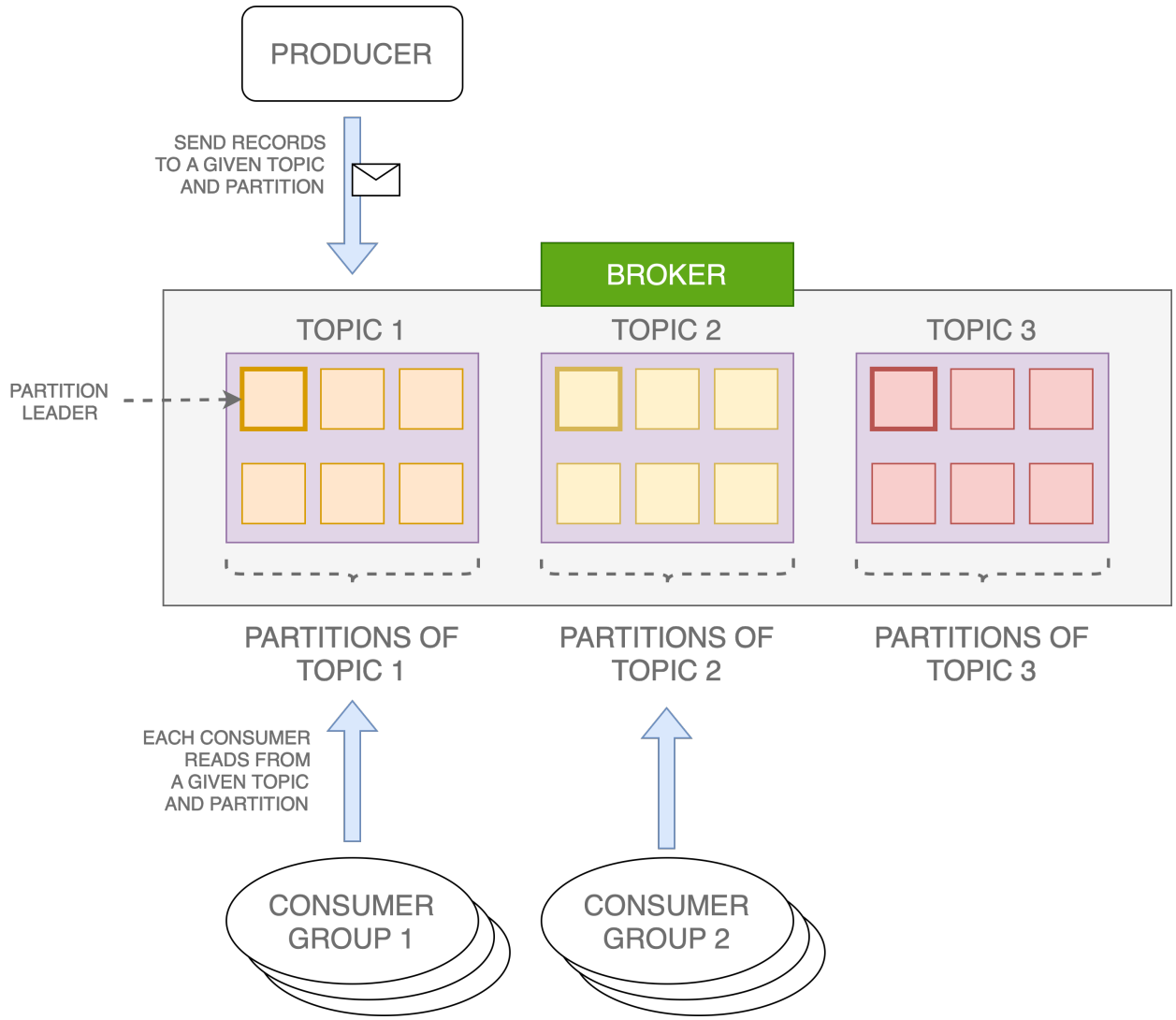
A **topic** in Kafka is a log of events. Logs are easy to understand, because they are simple data structures with well-known semantics. Topics have the following characteristics:

1. Append only: new messages are always written at the end of the log.
2. Topics can only be read by seeking an arbitrary offset in the log, then by scanning sequential log entries.
3. Events in the log are immutable.

Topics are defined server-side, in a default setting they cannot be created dynamically by a Kafka Producer but they must be created in advance.

Kafka Partitions

A Kafka topic can be divided into one or more **partitions**, and the number of partitions per topic is configurable. Partitioning takes the single topic log and breaks it into multiple logs, each of which can live on a separate node in the Kafka cluster. This way, the work of storing messages, writing new messages, and processing existing messages can be split among many nodes in the cluster. Out of that, one partition will be the **leader**. All read and write to a topic goes through the leader, and the leader coordinates updating the replicas. If a leader fails, a replica in the list is assigned as the new leader.



Producers, Consumers and Consumer Groups

In Kafka, each topic is divided into a set of logs known as partitions. **Producers** write to the tail of these logs and **consumers** read the logs at their own pace. Kafka scales topic consumption by distributing partitions among a **consumer group**, which is a set of consumers sharing a common group identifier. Each partition in the topic is assigned to **exactly one** member in the group, so there will be no two consumers reading from the same partition inside the same group. When all partitions are assigned, there might be some idle consumers.

Cloud Stack Configurations

Endpoint Configuration

Version: 1.0.0

The `KafkaClientCloudEndpoint` layer allows defining properties that dictate the encoding of the payload and whether to compress it using GZIP format.

If using the `RAW` encoding, **only** the body of the `KuraPayload` is serialized, whereas the metrics and position data are not included in the record. Instead, when using `Simple JSON` or `Kura Protobuf` all the data present in the `KuraPayload` is encoded. Whether to include such data is controlled by the Publisher Configuration properties.

Attention

In order to use `Kura Protobuf` some service should expose the following interfaces:

- `org.eclipse.kura.cloud.CloudPayloadProtoBufEncoder`, and
- `org.eclipse.kura.cloud.CloudPayloadProtoBufDecoder`

If running on ESF < 7.2.0, a default `org.eclipse.kura.cloud.CloudService` provides such interfaces.

Data Service Layer Configuration

Version: 1.0.0

Data service configurations are the canonical ones defined for other ESF Cloud Connections. Please refer to the ESF documentation, Data Service Configuration section *(https://esf.eurotech.com/docs/data-service)*.

Data Transport Layer configuration

Data Transport Layer Configuration

Version: 1.0.0

The `KafkaClientDataTransport` layer allows defining the configuration for the underlying `KafkaConsumer` and `KafkaProducer` instances that are used in this Cloud Connection. The user should have a good understanding of how a Kafka Client can be configured. Wrong configuration combinations can lead the producer or the consumer to throw exceptions. The configurable properties that are shown in the ESF UI can be further referenced in the Kafka Producer configurations documentation *(https://kafka.apache.org/documentation/#producerconfigs)* and the Kafka Consumer configurations documentation *(https://kafka.apache.org/documentation/#consumerconfigs)*.

Attention

The Data Transport layer creates the Kafka Producer and Kafka Consumer instances with the most secure setting available: **SSL authentication**. This option requires specifying a valid Keystore Path, otherwise, the Data Transport layer will not be initialized. Section Security configurations explains in more detail the security-related configurations and their setup.

Producer-related configurations

Version: 1.0.0

The producer consists of a pool of buffer space that holds records that haven't yet been transmitted to the server as well as a background I/O thread that is responsible for turning these records into requests and transmitting them to the cluster.

Each cloud connection instantiates a single `KafkaProducer`, which is shared among the CloudPublishers.

> **Attention**
>
> The producer implemented in this layer is not transactional nor idempotent.

Acknowledgements

The acks config controls the criteria under which requests are considered complete. The `ALL` setting will result in blocking on the full commit of the record, the slowest but most durable setting.

When this property is set to `ALL` and the oldest in-flight message is deleted from the data service, the acknowledgement might never be received. This causes the producer to never send new messages. Hence, when using this setting it is recommended tweaking the `max.inflight.requests.per.connection` config, the QoS of the produced message (Publisher configurations), or the store settings (DataService layer).

Default configurations

The instantiated `KafkaProducer` is initialized with some default configurations that are not definable by the user:

- `retries=0`: the producer will not try to resend any record whose send fails with a potentially transient error.
- `linger.ms=0`: the producer groups together any records that arrive in between request transmissions into a single batched request. Normally this occurs only under load when records arrive faster than they can be sent out. With this setting to `0`, the client will not wait any amount of delay for accumulating larger batches of records. The accumulated records are always sent immediately.
- `request.timeout.ms=1000`: maximum amount of time the client will wait for the response of a request.

- `delivery.timeout.ms=1000`: upper bound on the time to report success or failure after send. This limits the total time that a record will be delayed prior to sending and the time to await acknowledgement from the broker.
- `key.serializer=org.apache.kafka.common.serialization.StringSerializer`: the producer is able to process only records with keys of type `java.lang.String`.
- `value.serializer=org.apache.kafka.common.serialization.ByteArraySerializer`: the producer is able to process only records with values of type `byte[]`.

All the other configs are defaulted according to the Kafka Producer configurations documentation *(https://kafka.apache.org/documentation/#producerconfigs)*.

Consumer-related configurations

Version: 1.0.0

Each cloud connection instantiates a single `KafkaConsumer`, which is shared among the CloudSubscribers.

Group management

A Kafka Consumer Group has the following properties:

- All the Consumers in a group have the same `group.id`.
- Only one Consumer reads each partition in the topic.
- The maximum number of `KafkaConsumers` is equal to the number of partitions in the topic. If there are more consumers than partitions, then some of the consumers will remain idle.
- A Consumer can read from more than one partition.

Since each cloud connection instantiates a single `KafkaConsumer`, there might be the need to create multiple cloud connections to leverage partitions assignments.

Consumer liveness settings

After subscribing to a set of topics, the consumer will automatically join the group when the Cloud Connection is connected. As long as the Cloud Connection is connected, the consumer will stay in the group and continue to receive messages from the partitions it was assigned to. Underneath the covers, the consumer sends periodic heartbeats to the server, whose frequency is controlled by `heartbeat.interval.ms` parameter.

If the consumer crashes or is unable to send heartbeats for a duration of `session.timeout.ms`, then the consumer will be considered dead and its partitions will be reassigned.

Avoiding livelocks

It is also possible that the consumer could encounter a "livelock" situation where it is continuing to send heartbeats, but no progress is being made. The setting `max.poll.interval.ms` allows to prevent the consumer from holding onto its partitions indefinitely in this case.

If no message is being sent inside the configured max interval, then the client will proactively leave the group so that another consumer can take over its partitions. When this happens, you

may see an offset commit failure. This is a safety mechanism which guarantees that only active members of the group are able to commit offsets.

Offset management

The consumer offset is a way of tracking the sequential order in which messages are received by Kafka topics. The Kafka consumer offset allows processing to continue from where it last left off if the connection is lost or if there is an unexpected failure.

Initially, when a Kafka consumer starts for a new topic, the offset begins at zero (`0`). The `auto.offset.reset` config kicks in **only** if the consumer group does not have a valid committed offset. Two scenarios might be possible:

1. A Cloud Connection is configured with the consumer group to be `group1`, `enable.auto.commit` is enabled, and subscribers have consumed 5 messages before disconnecting. Next time the Cloud Connection is reconnected, the `KafkaConsumer` will not use the `auto.offset.reset` config but will continue from the latest committed record.
2. A Cloud Connection is configured with the consumer group to be `group2` and no subscribers have consumed messages yet. There is no offset stored anywhere and this time the `auto.offset.reset` config will decide whether to start from the beginning of the topic (`EARLIEST`) or from the end of the topic (`LATEST`).

Setting `enable.auto.commit` means that offsets are committed automatically with a frequency controlled by the config `auto.commit.interval.ms`. If not set, when the consumer gets disconnected, the `auto.offset.reset` policy is applied since the consumed records are not committed.

Partitioning

Kafka dynamically assign a fair share of the partitions for those topics based on the active consumers in the group. This version of `KafkaClientCloudConnection` does not allow defining the partitioning of the `KafkaConsumer`.

Default configurations

The instantiated `KafkaConsumer` is initialized with some default configurations that are not definable by the user:

- `key.deserializer=org.apache.kafka.common.serialization.StringDeserializer`: the consumer is able to process only records with keys of type `java.lang.String`.
- `value.deserializer=org.apache.kafka.common.serialization.ByteArrayDeserializ` the consumer is able to process only records with values of type `byte[]`.

All the other configs are defaulted according to the Kafka Consumer configurations documentation *(https://kafka.apache.org/documentation/#consumerconfigs)*.

Security configurations

Version: 1.0.0

This section details the various security configurations that are possible to apply at the transport level. The `security.protocol` field allows to apply the following security mechanisms (in order of increased security):

- `PLAINTEXT`: no encryption is made.
- `SASL/PLAINTEXT`: username/password-based authentication using Simple Authentication and Security Layer (SASL). Using this option **no encryption** is made, hence the credentials and the data are exchanged in cleartext.
- `SASL/PLAINTEXT over TLS`: username/password-based authentication using Simple Authentication and Security Layer (SASL). The encryption is made at transport level using TLS.
- `SSL`: SSL-based two-way authentication. This is the most secure setting and the **default one**. Please read the following sections to understand how to setup SSL authentication.

The following sections detail how to set up SSL and SASL authentication.

Attention

Configuring the cloud connector to use a security protocol that is not advertised on the broker will lead to a `java.lang.OutOfMemoryError: Java heap space`. This is a known Kafka issue, refer to Apache Issue 4090 *(https://issues.apache.org/jira/browse/KAFKA-4090)* for more details. Hence, be careful to match the `bootstrap.servers` configuration with the `security.protocol` one. In case the error happens, it is sufficient to click on the "Connect/Disconnect" button of the cloud connection, correct the configuration, and then click again on "Connect/Disconnect".

Configure SSL two-way authentication

The Kafka Cloud Connector requires a keystore where to store the trusted certificates and the needed key pairs. ESF provides a *Keystore Configuration* service that can be used to create keystores and store certificates. Refer to Kafka Encryption and Authentication using SSL *(https://kafka.apache.org/documentation.html#security_ssl)* for further details.

1. Create test certificates

This section explains how to create a server and a client certificate signed by a CA. The server certificate is created so to allow hostname verification by the clients.

Certification Authority CA

From the machine running the test broker, create a x509 configuration file named `openssl-ca.cnf` from the following template:

```
HOME = .
RANDFILE = $ENV::HOME/.rnd

####################################################################
[ ca ]
default_ca = CA_default # The default ca section

[ CA_default ]

base_dir = .
certificate = $base_dir/cacert.pem # The CA certifcate
private_key = $base_dir/cakey.pem # The CA private key
new_certs_dir = $base_dir # Location for new certs after signing
database = $base_dir/index.txt # Database index file
serial = $base_dir/serial.txt # The current serial number

default_days = 1000 # How long to certify for
default_crl_days = 30 # How long before next CRL
default_md = sha256 # Use public key default MD
preserve = no # Keep passed DN ordering

x509_extensions = ca_extensions # The extensions to add to the cert

email_in_dn = no # Don't concat the email in the DN
copy_extensions = copy # Required to copy SANs from CSR to cert

####################################################################
[ req ]
default_bits = 4096
default_keyfile = cakey.pem
distinguished_name = ca_distinguished_name
x509_extensions = ca_extensions
string_mask = utf8only

####################################################################
[ ca_distinguished_name ]
countryName = countryName
countryName_default = US

stateOrProvinceName = stateOrProvinceName
stateOrProvinceName_default = DEFAULT
```

```
localityName = Amaro
localityName_default = DEFAULT

organizationName = Eurotech
organizationName_default = DEFAULT

organizationalUnitName = ESF
organizationalUnitName_default = DEFAULT

commonName = <ipbroker>
commonName_default = IP:<ip-broker>

emailAddress = test@test.com
emailAddress_default = test@test.com

#####################################################################
[ ca_extensions ]

subjectKeyIdentifier = hash
authorityKeyIdentifier = keyid:always, issuer
basicConstraints = critical, CA:true
keyUsage = keyCertSign, cRLSign

#####################################################################
[ signing_policy ]
countryName = optional
stateOrProvinceName = optional
localityName = optional
organizationName = optional
organizationalUnitName = optional
commonName = supplied
emailAddress = optional

#####################################################################
[ signing_req ]
subjectKeyIdentifier = hash
authorityKeyIdentifier = keyid,issuer
basicConstraints = CA:FALSE
keyUsage = digitalSignature, keyEncipherment
```

Next, generate a Certification Authority key pair using the previously created configuration file:

```
echo 01 > serial.txt
touch index.txt
openssl req -x509 -config openssl-ca.cnf -newkey rsa:4096 -sha256 -nc
```

This command will create a CA private key `cakey.pem` and the CA public certificate `cacert.pem`. `serial.txt` and `index.txt` are used by the tool when multiple signatures are issued. Import the certificate into the truststore:

```
keytool -keystore kafka.server.truststore.jks -alias CARoot -import -
```

Broker certificate

Create a broker certificate to be signed later on with the CA:

```
keytool -keystore kafka.server.keystore.jks -alias localhost -validi
keytool -list -v -keystore kafka.server.keystore.jks
```

This generates a new keystore `kafka.server.keystore.jks` in PKCS12 format which contains a new key pair with alias `localhost`. The option `-ext SAN=IP:<ip-broker>` allows to include in the generated certificate the Subject Alternative Name, which can be used by the client for verifying the hostname.

Create a Certificate Signature Request by specifying the SAN for hostname verification:

```
keytool -keystore kafka.server.keystore.jks -alias localhost -certre
```

Create a signed server certificate from the previously created CSR:

```
openssl ca -config openssl-ca.cnf -policy signing_policy -extensions
openssl x509 -in servercert-signed.pem -text -noout
```

Import the CA public certificate and the signed server public certificate into the server keystore:

```
keytool -keystore kafka.server.keystore.jks -alias CARoot -import -f
keytool -keystore kafka.server.keystore.jks -alias localhost -import
keytool -list -v -keystore kafka.server.keystore.jks
```

Client key pair

Generate a private key:

```
openssl genrsa -out client.key 4096
```

Create a CSR:

```
openssl req -new -key client.key -out client.csr
```

Sign it with the CA's key pair:

```
openssl x509 -req -CA cacert.pem -CAkey cakey.pem -in client.csr -ou
```

Export the `client.key` private key in PKCS8 format for usage in ESF:

```
openssl pkcs8 -topk8 -inform PEM -outform PEM -in client.key -out cl
```

2. Configure kafka broker

The broker can be configured with the following properties (usually found in `server.properties` file):

```
ssl.truststore.location=<path/to/kafka.server.truststore.jks>
ssl.truststore.password=<password>
ssl.keystore.location=<path/to/kafka.server.keystore.jks>
ssl.keystore.password=<password>
ssl.key.password=<password>
listeners=PLAINTEXT://<ip-broker>:9092,SSL://<ip-broker>:9093
advertised.listeners=PLAINTEXT://<ip-broker>:9092,SSL://<ip-broker>:9
ssl.client.auth=required
ssl.enabled.protocols=TLSv1.2
ssl.endpoint.identification.algorithm=
```

This allows the broker to listen for unencrypted connections on `<ip-broker>:9092` and requires SSL on `<ip-broker>:9093`. Configure the correct TLS protocol and leave `ssl.endpoint.identification.algorithm=` empty for disabling hostname verification by the broker.

3. Create a new keystore (optional)

In this tutorial, a new keystore specific for the Kafka client will be created. An existing keystore might be used as well. In the ESF UI, move to "Keystore Configuration" under the "Security" section. Add a new keystore specifying a file path and a password.

4. Import client key pair and CA public certificate in ESF

This can be done by adding the key pair and the trusted certificate from the ESF UI in the "Security" section. Make sure to select the previously created keystore.

5. Configure Kafka transport

In the `KafkaClientDataTransport` layer, set `security.protocol` to `SSL` and configure the `Keystore Path` as the path defined at step 3. Set the `Enable hostname verification` option to `true` to enable hostname verification. The certificates created in step 1 should enable this feature.

Configure SASL authentication

The Apache Kafka® Cloud Connector supports authentication via Simple Authentication and Security Layer (SASL). The `security.protocol` field allows to choose from two different SASL mechanisms:

- simple `SASL/PLAINTEXT`, and
- `SASL/PLAINTEXT over TLS`.

`SASL/PLAINTEXT` is a simple username/password authentication-based mechanism. Note that it is not encrypting the underlying connection, so the sent credentials are in clear. This option is intended to use mainly for debug purposes when the connecting Kafka broker is not securely configured. The `SASL/PLAINTEXT` over TLS option uses a username/password authentication-based mechanism but the connection is encrypted at transport level, using TLS. This option is safer than the previous.

The next sections are guides to configuring the Kafka broker and the cloud connector to use SASL authentication mechanisms. Refer to the Confluent documentation: Authentication with SASL using JAAS *(https://docs.confluent.io/platform/current/kafka/authentication_sasl/index.html)* or Kafka Authentication using SASL *(https://kafka.apache.org/documentation.html#security_sasl)* for further details.

1. Configure the Kafka broker for SASL-PLAINTEXT

To configure the broker, edit `server.properties` file by adding `PLAIN` to the enabled SASL mechanisms:

```
sasl.enabled.mechanisms=PLAIN
```

Configure the broker to listen for SASL connections by adding the following listeners in the `server.properties` file:

```
listeners=SASL_PLAINTEXT://<broker-ip>:<port>
advertised.listeners=SASL_PLAINTEXT://<broker-ip>:<port>
```

In case `SASL/PLAINTEXT over TLS` is configured, replace `SASL_PLAINTEXT` with `SASL_SSL` and configure the keystore and truststore as in the previous sections. Next, create a JAAS configuration file as follows:

```
KafkaServer {
 org.apache.kafka.common.security.plain.PlainLoginModule required
 username="<username>"
 password="<password>"
 user_<username>="<password>";
};
```

Insert as many `user_<username>=<password>` as many credentials are wanted. This configuration file is passed as a JVM parameter. Modify `bin/kafka-server-start.sh` to specify the security configuration in `EXTRA_ARGS`:

```
-Djava.security.auth.login.config=/path/to/jaas/file
```

At this point, the cloud connector should be able to connect to the broker with the specified credentials in `SASL username` and `SASL password`.

2.Configure SASL-PLAINTEXT over TLS

In case `SASL/PLAINTEXT over TLS` is used, the `Keystore Path` property must be set to the keystore containing the CA certificate for setting up the underlying SSL connection. The same keystore as the one used in the previous section can be used.

Publisher/Subscriber Configurations

Cloud Publisher Configuration

Version: 1.0.0

The `com.eurotech.framework.kafka.client.cloudconnection.publisher.KafkaClientPublishe` allows interaction with the `KafkaProducer` instance of the Cloud Connection. The configuration properties for the publisher are detailed below.

Events created by the underlying `KafkaProducer` are sent as `ProducerRecord`. Such records have a fixed type for the key and the value and are serialized as follows:

- **value**: is of type `byte[]` and is serialized using the default `org.apache.kafka.common.serialization.ByteArraySerializer` *(https://kafka.apache.org/0100/javadoc/org/apache/kafka/common/serialization/ByteArraySerializer.html)*.
- **key** (optional): is of type `java.lang.String` and is serialized using the default `org.apache.kafka.common.serialization.StringSerializer` *(https://kafka.apache.org/0100/javadoc/org/apache/kafka/common/serialization/StringSerializer.html)*.

Producer topic

This configuration is mandatory. It sets the Kafka topic to which send the created events. Valid characters for Kafka topics match the following pattern:

```
[a-zA-Z0-9\._\-]
```

However, one thing to keep in mind is that due to limitations in metric names, topics with a period (`.`) or underscore (_) could collide. To avoid issues it is best to use either, but not both.

Key

This configuration is optional. When specified, all the messages of this publisher are sent to a single partition. In Kafka, the messages are guaranteed to be processed in order only if they share the same key, so specify the key only if the application requires message ordering.

The default partitioner used by the producer in Kafka works as follows. The producer first calculates a numeric hash of the key using `murmur2` algorithm, and then selects the partition number by the following formula:

```
murmur2(key) % number_of_partitions
```

Where `%` is the modulus operation.

For example, the `murmur2` of the string `testkey` is 2871421366. If the topic has 10 partitions, `2871421366 % 10 = 6`. Hence, using the default partitioner, every message send with key `testkey` will land in partition number 6.

Publish metrics & Publish position

When set to `false`, the publisher will strip off such data from the `KuraPayload` before sending it. Metrics and position data are always ignored when the **Payload Encoding** property of the `KafkaClientCloudEndpoint` section is set to `RAW`.

Priority

Used by the data service, sets the priority of the message. This priority is used to determine the republishing order of the unpublished messages. A message reordering may occur.

Quality of service (QoS)

This parameter is used by the data service to guarantee that the message is delivered at most once (QoS = 0) or at least once (QoS > 0).

Cloud Subscriber Configuration

Version: 1.0.0

The `com.eurotech.framework.kafka.client.cloudconnection.subscriber.KafkaClientSubscri` allows interaction with the `KafkaConsumer` instance of the Cloud Connection. The configuration properties for the subscriber are detailed below.

Events received by the underlying `KafkaConsumer` are of type `ProducerRecord`. Such records have a fixed type for the key and the value and are deserialized as follows:

- **value**: is of type `byte[]` and is deserialized using the default `org.apache.kafka.common.serialization.ByteArrayDeserializer` *(https://kafka.apache.org/0100/javadoc/org/apache/kafka/common/serialization/ByteArrayDeserializer.html)*.
- **key**: is of type `java.lang.String` and is deserialized using the default `org.apache.kafka.common.serialization.StringDeserializer` *(https://kafka.apache.org/0100/javadoc/org/apache/kafka/common/serialization/StringDeserializer.html)*.

Topic

Required field, sets the topic to which subscribe. Valid characters for Kafka topics match the following pattern:

```
[a-zA-Z0-9\._\-]
```

However, one thing to keep in mind is that due to limitations in metric names, topics with a period (`.`) or underscore (_) could collide. To avoid issues it is best to use either, but not both.

References

References

ESF Documentation

- Cloud Services *(https://esf.eurotech.com/docs/cloud-services)*
- Data Service Configuration *(https://esf.eurotech.com/docs/data-service)*
- Multi-cloud Connection Support *(https://esf.eurotech.com/docs/overview)*

Kafka Documentation

- Kafka Introduction *(https://kafka.apache.org/intro)*
- Kafka Producer configurations *(https://kafka.apache.org/documentation/#producerconfigs)*
- Kafka Consumer configurations *(https://kafka.apache.org/documentation/#consumerconfigs)*
- KafkaProducer API *(https://docs.confluent.io/platform/current/clients/javadocs/javadoc/org/apache/kafka/clients/producer/KafkaProducer.html)*
- KafkaConsumer API *(https://kafka.apache.org/22/javadoc/org/apache/kafka/clients/consumer/KafkaConsumer.html)*
- Apache Kafka Homepage *(https://kafka.apache.org)*
- Kafka Encryption and Authentication using SSL *(https://kafka.apache.org/documentation.html#security_ssl)*
- Kafka Authentication using SASL *(https://kafka.apache.org/documentation.html#security_sasl)*
- Confluent documentation: Authentication with SASL using JAAS *(https://docs.confluent.io/platform/current/kafka/authentication_sasl/index.html)*

Kafka Known Issues

- Apache Issue 4090 *(https://issues.apache.org/jira/browse/KAFKA-4090)*