Red Hat AMQ Streams 2.6

Developing Kafka client applications

Develop client applications to interact with Kafka using AMQ Streams

Last Updated: 2024-04-12

Red Hat AMQ Streams 2.6 Developing Kafka client applications

Develop client applications to interact with Kafka using AMQ Streams

Legal Notice

Abstract

Develop client applications that can send and receive messages through Kafka brokers. Set up secure access between the clients and the brokers.

Table of Contents

MAKING OPEN SOURCE MORE INCLUSIVE

Red Hat is committed to replacing problematic language in our code, documentation, and web properties. We are beginning with these four terms: master, slave, blacklist, and whitelist. Because of the enormity of this endeavor, these changes will be implemented gradually over several upcoming releases. For more details, see our CTO Chris Wright's message.

CHAPTER 1. DEVELOPING CLIENTS OVERVIEW

Develop Kafka client applications for your AMQ Streams installation that can produce messages, consume messages, or do both. You can develop client applications for use with AMQ Streams on OpenShift or AMQ Streams on RHEL.

Messages comprise an optional key and a value that contains the message data, plus headers and related metadata. The key identifies the subject of the message, or a property of the message. You must use the same key if you need to process a group of messages in the same order as they are sent.

Messages are delivered in batches. Messages contain headers and metadata that provide details that are useful for filtering and routing by clients, such as the timestamp and offset position for the message.

Kafka provides client APIs for developing client applications. Kafka producer and consumer APIs are the primary means of interacting with a Kafka cluster in a client application. The APIs control the flow of messages. The producer API sends messages to Kafka topics, while the consumer API reads messages from topics.

AMQ Streams supports clients written in Java. How you develop your clients depends on your specific use case. Data durability might be a priority or high throughput. These demands can be met through configuration of your clients and brokers. All clients, however, must be able to connect to all brokers in a given Kafka cluster.

1.1. SUPPORTING A HTTP CLIENT

As an alternative to using the Kafka producer and consumer APIs in your client, you can set up and use the AMQ Streams Kafka Bridge. The Kafka Bridge provides a RESTful interface that allows HTTP-based clients to interact with a Kafka cluster. It offers the advantages of a web API connection to Strimzi, without the need for client applications that need to interpret the Kafka protocol. Kafka uses a binary protocol over TCP.

For more information, see Using the AMQ Streams Kafka Bridge .

1.2. TUNING YOUR PRODUCERS AND CONSUMERS

You can add more configuration properties to optimize the performance of your Kafka clients. You probably want to do this when you've had some time to analyze how your client and broker configuration performs.

For more information, see Kafka configuration tuning.

1.3. MONITORING CLIENT INTERACTION

Distributed tracing facilitates the end-to-end tracking of messages. You can enable tracing in Kafka consumer and producer client applications.

For more information, see the documentation for distributed tracing in the following guides:

- Deploying and Upgrading AMQ Streams on OpenShift
- Using AMQ Streams on RHEL

NOTE

When we use the term client application, we're specifically referring to applications that use Kafka producers and consumers to send and receive messages to and from a Kafka cluster. We are not referring to other Kafka components, such as Kafka Connect or Kafka Streams, which have their own distinct use cases and functionality.

CHAPTER 2. CLIENT DEVELOPMENT PREREQUISITES

The following prerequisites are required for developing clients to use with AMQ Streams.

- You have a Red Hat account.
- You have a Kafka cluster running in AMQ Streams.
- Kafka brokers are configured with listeners for secure client connections.
- Topics have been created for your cluster.
- You have an IDE to develop and test your client.
- JDK 11 or later is installed.

CHAPTER 3. ADDING CLIENT DEPENDENCIES TO YOUR MAVEN PROJECT

If you are developing Java-based Kafka clients, you can add the Red Hat dependencies for Kafka clients, including Kafka Streams, to the **pom.xml** file of your Maven project. Only client libraries built by Red Hat are supported for AMQ Streams.

You can add the following artifacts as dependencies:

kafka-clients

Contains the Kafka **Producer**, **Consumer**, and **AdminClient** APIs.

- The **Producer** API enables applications to send data to a Kafka broker.
- The **Consumer** API enables applications to consume data from a Kafka broker.
- The **AdminClient** API provides functionality for managing Kafka clusters, including topics, brokers, and other components.

kafka-streams

Contains the **KafkaStreams** API.
Kafka Streams enables applications to receive data from one or more input streams. You can use this API to run a sequence of real-time operations on streams of data, like mapping, filtering, and joining. You can use Kafka Streams to write results into one or more output streams. It is part of the **kafka-streams** JAR package that is available in the Red Hat Maven repository.

3.1. ADDING A KAFKA CLIENTS DEPENDENCY TO YOUR MAVEN PROJECT

Add a Red Hat dependency for Kafka clients to your Maven project.

Prerequisites

- A Maven project with an existing **pom.xml**.

Procedure

1. Add the Red Hat Maven repository to the **<repositories>** section of the **pom.xml** file of your Maven project.

```
<repositories>
  <repository>
    <id>redhat-maven</id>
    <url>https://maven.repository.redhat.com/ga/</url>
  </repository>
</repositories>
```

2. Add **kafka-clients** as a **<dependency>** to the **pom.xml** file of your Maven project.

```
<dependencies>
  <dependency>
    <groupId>org.apache.kafka</groupId>
```

```
        <artifactId>kafka-clients</artifactId>
        <version>3.6.0.redhat-00005</version>
    </dependency>
</dependencies>
```

3. Build the Maven project to add the Kafka client dependency to the project.

3.2. ADDING A KAFKA STREAMS DEPENDENCY TO YOUR MAVEN PROJECT

Add a Red Hat dependency for Kafka Streams to your Maven project.

Prerequisites

- A Maven project with an existing **pom.xml**.

Procedure

1. Add the Red Hat Maven repository to the **<repositories>** section of the **pom.xml** file of your Maven project.

```
<repositories>
    <repository>
        <id>redhat-maven</id>
        <url>https://maven.repository.redhat.com/ga/</url>
    </repository>
</repositories>
```

2. Add **kafka-streams** as a **<dependency>** to the **pom.xml** file of your Maven project.

```
<dependencies>
    <dependency>
        <groupId>org.apache.kafka</groupId>
        <artifactId>kafka-streams</artifactId>
        <version>3.6.0.redhat-00005</version>
    </dependency>
</dependencies>
```

3. Build the Maven project to add the Kafka Streams dependency to the project.

3.3. ADDING AN OAUTH 2.0 DEPENDENCY TO YOUR MAVEN PROJECT

Add a Red Hat dependency for OAuth 2.0 to your Maven project.

Prerequisites

- A Maven project with an existing **pom.xml**.

Procedure

1. Add the Red Hat Maven repository to the **<repositories>** section of the **pom.xml** file of your Maven project.

```
<repositories>
    <repository>
        <id>redhat-maven</id>
        <url>https://maven.repository.redhat.com/ga/</url>
    </repository>
</repositories>
```

2. Add **kafka-oauth-client** as a **<dependency>** to the **pom.xml** file of your Maven project.

```
<dependency>
 <groupId>io.strimzi</groupId>
 <artifactId>kafka-oauth-client</artifactId>
 <version>0.14.0.redhat-00006</version>
</dependency>
```

3. Build the Maven project to add the OAuth 2.0 dependency to the project.

CHAPTER 4. CONFIGURING CLIENT APPLICATIONS FOR CONNECTING TO A KAFKA CLUSTER

To connect to a Kafka cluster, a client application must be configured with a minimum set of properties that identify the brokers and enable a connection. Additionally, you need to add a serializer/deserializer mechanism to convert messages into or out of the byte array format used by Kafka. When developing a consumer client, you begin by adding an initial connection to your Kafka cluster, which is used to discover all available brokers. When you have established a connection, you can begin consuming messages from Kafka topics or producing messages to them.

Although not required, a unique client ID is recommended so that you can identity your clients in logs and metrics collection.

You can configure the properties in a properties file. Using a properties file means you can modify the configuration without recompiling the code.

For example, you can load the properties in a Java client using the following code:

Loading configuration properties into a client

```
Properties props = new Properties();
try (InputStream propStream = Files.newInputStream(Paths.get(filename))) {
    props.load(propStream);
}
```

You can also use add the properties directly to the code in a configuration object. For example, you can use the **setProperty()** method for a Java client application. Adding properties directly is a useful option when you only have a small number of properties to configure.

4.1. BASIC PRODUCER CLIENT CONFIGURATION

When you develop a producer client, configure the following:

- A connection to your Kafka cluster
- A serializer to transform message keys into bytes for the Kafka broker
- A serializer to transform message values into bytes for the Kafka broker

You might also add a compression type in case you want to send and store compressed messages.

Basic producer client configuration properties

```
client.id = my-producer-id (1)
bootstrap.servers = my-cluster-kafka-bootstrap:9092 (2)
key.serializer = org.apache.kafka.common.serialization.StringSerializer (3)
value.serializer = org.apache.kafka.common.serialization.StringSerializer (4)
```

1. The logical name for the client.
2. Bootstrap address for the client to be able to make an initial connection to the Kafka cluster.
3. Serializer to transform message keys into bytes before being sent to the Kafka broker.

4 Serializer to transform message values into bytes before being sent to the Kafka broker.

Adding producer client configuration directly to the code

```
Properties props = new Properties();
props.setProperty(ProducerConfig.CLIENT_ID_CONFIG, "my-producer-id");
props.setProperty(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "my-cluster-kafka-
bootstrap:9092");
props.setProperty(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
StringSerializer.class.getName());
props.setProperty(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
StringSerializer.class.getName());
KafkaProducer<String, String> producer = new KafkaProducer<>(properties);
```

The **KafkaProducer** specifies string key and value types for the messages it sends. The serializers used must be able to convert the key and values from the specified type into bytes before sending them to Kafka.

4.2. BASIC CONSUMER CLIENT CONFIGURATION

When you develop a consumer client, configure the following:

- A connection to your Kafka cluster
- A deserializer to transform the bytes fetched from the Kafka broker into message keys that can be understood by the client application
- A deserializer to transform the bytes fetched from the Kafka broker into message values that can be understood by the client application

Typically, you also add a consumer group ID to associate the consumer with a consumer group. A consumer group is a logical entity for distributing the processing of a large data stream from one or more topics to parallel consumers. Consumers are grouped using a **group.id**, allowing messages to be spread across the members. In a given consumer group, each topic partition is read by a single consumer. A single consumer can handle many partitions. For maximum parallelism, create one consumer for each partition. If there are more consumers than partitions, some consumers remain idle, ready to take over in case of failure.

Basic consumer client configuration properties

```
client.id = my-consumer-id 1
group.id = my-group-id 2
bootstrap.servers = my-cluster-kafka-bootstrap:9092 3
key.deserializer = org.apache.kafka.common.serialization.StringDeserializer 4
value.deserializer = org.apache.kafka.common.serialization.StringDeserializer 5
```

1 The logical name for the client.

2 A group ID for the consumer to be able to join a specific consumer group.

3 Bootstrap address for the client to be able to make an initial connection to the Kafka cluster.

4 Deserializer to transform the bytes fetched from the Kafka broker into message keys.

5 Deserializer to transform the bytes fetched from the Kafka broker into message values.

Adding consumer client configuration directly to the code

```
Properties props = new Properties();
props.setProperty(ConsumerConfig.CLIENT_ID_CONFIG, "my-consumer-id");
props.setProperty(ConsumerConfig.GROUP_ID_CONFIG, "my-group-id");
props.setProperty(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "my-cluster-kafka-bootstrap:9092");
props.setProperty(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class.getName());
props.setProperty(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class.getName());
KafkaConsumer<String, String> consumer = new KafkaConsumer<>(properties);
```

The **KafkaConsumer** specifies string key and value types for the messages it receives. The serializers used must be able to convert the bytes received from Kafka into the specified types.

NOTE

Each consumer group must have a unique **group.id**. If you restart a consumer with the same **group.id**, it resumes consuming messages from where it left off before it was stopped.

CHAPTER 5. CONFIGURING SECURE CONNECTIONS

Securing the connection between a Kafka cluster and a client application helps to ensure the confidentiality, integrity, and authenticity of the communication between the cluster and the client.

To achieve a secure connection, you can introduce configuration related to authentication, encryption, and authorization:

Authentication

Use an authentication mechanism to verify the identity of a client application.

Encryption

Enable encryption of data in transit between the client and broker using SSL/TLS encryption.

Authorization

Control client access and operations allowed on Kafka brokers based on the authenticated identity of a client application.

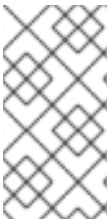
Authorization cannot be used without authentication. If authentication is not enabled, it's not possible to determine the identity of clients, and therefore, it's not possible to enforce authorization rules. This means that even if authorization rules are defined, they will not be enforced without authentication.

In AMQ Streams, listeners are used to configure the network connections between the Kafka brokers and the clients. Listener configuration options determine how the brokers listen for incoming client connections and how secure access is managed. The exact configuration required depends on the authentication, encryption, and authorization mechanisms you have chosen.

You configure your Kafka brokers and client applications to enable security features. The general outline to secure a client connection to a Kafka cluster is as follows:

1. Install the AMQ Streams components, including the Kafka cluster.
2. For TLS, generate TLS certificates for each broker and client application.
3. Configure listeners in the broker configuration for secure connection.
4. Configure the client application for secure connection.

Configure your client application according to the mechanisms you are using to establish a secure and authenticated connection with the Kafka brokers. The authentication, encryption, and authorization used by a Kafka broker must match those used by a connecting client application. The client application and broker need to agree on the security protocols and configurations for secure communication to take place. For example, a Kafka client and the Kafka broker must use the same TLS versions and cipher suites.



NOTE

Mismatched security configurations between the client and broker can result in connection failures or potential security vulnerabilities. It's important to carefully configure and test both the broker and client application to ensure they are properly secured and able to communicate securely.

5.1. SETTING UP BROKERS FOR SECURE ACCESS

Before you can configure client applications for secure access, you must first set up the brokers in your Kafka cluster to support the security mechanisms you want to use. To enable secure connections, you create listeners with the appropriate configuration for the security mechanisms.

5.1.1. Establishing a secure connection to a Kafka cluster running on RHEL

When using AMQ Streams on RHEL, the general outline to secure a client connection to a Kafka cluster is as follows:

1. Install the AMQ Streams components, including the Kafka cluster, on the RHEL server.
2. For TLS, generate TLS certificates for all brokers in the Kafka cluster.
3. Configure listeners in the broker configuration properties file.
    - Configure authentication for your Kafka cluster listeners, such as TLS or SASL SCRAM-SHA-512.
    - Configure authorization for all enabled listeners on the Kafka cluster, such as **simple** authorization.
4. For TLS, generate TLS certificates for each client application.
5. Create a **config.properties** file to specify the connection details and authentication credentials used by the client application.
6. Start the Kafka client application and connect to the Kafka cluster.
    - Use the properties defined in the **config.properties** file to connect to the Kafka broker.
7. Verify that the client can successfully connect to the Kafka cluster and consume and produce messages securely.

For more information on setting up your brokers, see Using AMQ Streams on RHEL .

5.1.2. Configuring secure listeners for a Kafka cluster on RHEL

Use a configuration properties file to configure listeners in Kafka. To configure a secure connection for Kafka brokers, you set the relevant properties for TLS, SASL, and other security-related configurations in this file.

Here is an example configuration of a TLS listener specified in a **server.properties** configuration file for a Kafka broker, with a keystore and truststore in PKCS#12 format:

Example listener configuration in server.properties

```
listeners = listener_1://0.0.0.0:9093, listener_2://0.0.0.0:9094
listener.security.protocol.map = listener_1:SSL, listener_2:PLAINTEXT
ssl.keystore.type = PKCS12
ssl.keystore.location = /path/to/keystore.p12
ssl.keystore.password = <password>
ssl.truststore.type = PKCS12
ssl.truststore.location = /path/to/truststore.p12
ssl.truststore.password = <password>
```

```
ssl.client.auth = required
authorizer.class.name = kafka.security.auth.SimpleAclAuthorizer.
super.users = User:superuser
```

The **listeners** property specifies each listener name, and the IP address and port that the broker listens on. The protocol map tells the **listener_1** listener to use the SSL protocol for clients that use TLS encryption. **listener_2** provides PLAINTEXT connections for clients that do not use TLS encryption. The keystore contains the broker's private key and certificate. The truststore contains the trusted certificates used to verify the identity of the client application. The **ssl.client.auth** property enforces client authentication.

The Kafka cluster uses simple authorization. The authorizer is set to **SimpleAclAuthorizer**. A single super user is defined for unconstrained access on all listeners. AMQ Streams supports the Kafka **SimpleAclAuthorizer** and custom authorizer plugins.

If we prefix the configuration properties with **listener.name.<name_of_listener>**, the configuration is specific to that listener.

This is just a sample configuration. Some configuration options are specific to the type of listener. If you are using OAuth 2.0 or Open Policy Agent (OPA), you must also configure access to the authorization server or OPA server in a specific listener. You can create listeners based on your specific requirements and environment.

For more information on listener configuration, see the Apache Kafka documentation.

Using ACLs to fine-tune access

You can use Access Control Lists (ACLs) to fine-tune access to the Kafka cluster. To create and manage Access Control Lists (ACLs), use the **kafka-acls.sh** command line tool. The ACLs apply access rules to client applications.

In the following example, the first ACL grants read and describe permissions for a specific topic named **my-topic**. The **resource.patternType** is set to **literal**, which means that the resource name must match exactly.

The second ACL grants read permissions for a specific consumer group named **my-group**. The **resource.patternType** is set to **prefix**, which means that the resource name must match the prefix.

Example ACL configuration

```
bin/kafka-acls.sh --authorizer-properties zookeeper.connect=localhost:2181 --add \
--allow-principal User:my-user --operation Read --operation Describe --topic my-topic --resource-pattern-type literal \
--allow-principal User:my-user --operation Read --group my-group --resource-pattern-type prefixed
```

5.1.3. Establishing a secure connection to a Kafka cluster running on OpenShift

When using AMQ Streams on OpenShift, the general outline to secure a client connection to a Kafka cluster is as follows:

1. Use the Cluster Operator to deploy a Kafka cluster in your OpenShift environment. Use the **Kafka** custom resource to configure and install the cluster and create listeners.
    - Configure authentication for the listeners, such as TLS or SASL SCRAM-SHA-512. The Cluster Operator creates a secret that contains a cluster CA certificate to verify the identity of the Kafka brokers.

- Configure authorization for all enabled listeners, such as **simple** authorization.

2. Use the User Operator to create a Kafka user representing your client. Use the **KafkaUser** custom resource to configure and create the user.
   - Configure authentication for your Kafka user (client) that matches the authentication mechanism of a listener. The User Operator creates a secret that contains a client certificate and private key for the client to use for authentication with the Kafka cluster.
   - Configure authorization for your Kafka user (client) that matches the authorization mechanism of the listener. Authorization rules allow specific operations on the Kafka cluster.
3. Create a **config.properties** file to specify the connection details and authentication credentials required by the client application to connect to the cluster.
4. Start the Kafka client application and connect to the Kafka cluster.
   - Use the properties defined in the **config.properties** file to connect to the Kafka broker.
5. Verify that the client can successfully connect to the Kafka cluster and consume and produce messages securely.

For more information on setting up your brokers, see Configuring AMQ Streams on OpenShift.

5.1.4. Configuring secure listeners for a Kafka cluster on OpenShift

When you deploy a **Kafka** custom resource with AMQ Streams, you add listener configuration to the Kafka **spec**. Use the listener configuration to secure connections in Kafka. To configure a secure connection for Kafka brokers, set the relevant properties for TLS, SASL, and other security-related configurations at the listener level.

External listeners provide client access to a Kafka cluster from outside the OpenShift cluster. AMQ Streams creates listener services and bootstrap addresses to enable access to the Kafka cluster based on the configuration. For example, you can create external listeners that use the following connection mechanisms:

- Node ports
- loadbalancers
- Openshift routes

Here is an example configuration of a **nodeport** listener for a **Kafka** resource:

Example listener configuration in the Kafka resource

```
apiVersion: {KafkaApiVersion}
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
    listeners:
      - name: plaintext
        port: 9092
        type: internal
```

```
      tls: false
      configuration:
        useServiceDnsDomain: true
    - name: tls
      port: 9093
      type: internal
      tls: true
      authentication:
        type: tls
    - name: external
      port: 9094
      type: route
      tls: true
      authentication:
        type: tls
  authorization:
    type: simple
    superUsers:
      - CN=superuser
# ...
```

The **listeners** property is configured with three listeners: **plaintext**, **tls**, and **external**. The **external** listener is of type **nodeport**, and it uses TLS for both encryption and authentication. When you create the Kafka cluster with the Cluster Operator, CA certificates are automatically generated. You add cluster CA to the truststore of your client application to verify the identity of the Kafka brokers. Alternatively, you can configure AMQ Streams to use your own certificates at the broker or listener level. Using certificates at the listener level might be required when client applications require different security configurations. Using certificates at the listener level also adds an additional layer of control and security.

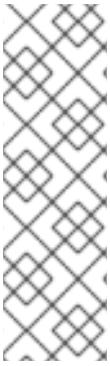
TIP

Use configuration provider plugins to load configuration data to producer and consumer clients. The configuration Provider plugin loads configuration data from secrets or ConfigMaps. For example, you can tell the provider to automatically get certificates from Strimzi secrets. For more information, see the AMQ Streams documentation for running onOpenShift.

The Kafka cluster uses simple authorization. The authorization property type is set to **simple**. A single super user is defined for unconstrained access on all listeners. AMQ Streams supports the Kafka **SimpleAclAuthorizer** and custom authorizer plugins.

This is just a sample configuration. Some configuration options are specific to the type of listener. If you are using OAuth 2.0 or Open Policy Agent (OPA), you must also configure access to the authorization server or OPA server in a specific listener. You can create listeners based on your specific requirements and environment.

For more information on listener configuration, see the **GenericKafkaListener** schema reference.



NOTE

When using a **route** type listener for client access to a Kafka cluster on OpenShift, the TLS passthrough feature is enabled. An OpenShift route is designed to work with the HTTP protocol, but it can also be used to proxy network traffic for other protocols, including the Kafka protocol used by Apache Kafka. The client establishes a connection to the route, and the route forwards the traffic to the broker running in the OpenShift cluster using the TLS Server Name Indication (SNI) extension to get the target hostname. The SNI extension allows the route to correctly identify the target broker for each connection.

Using ACLs to fine-tune access

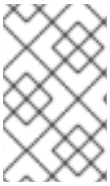
You can use Access Control Lists (ACLs) to fine-tune access to the Kafka cluster. To add Access Control Lists (ACLs), you configure the **KafkaUser** custom resource. When you create a **KafkaUser**, AMQ Streams automatically manages the creation and updates the ACLs. The ACLs apply access rules to client applications.

In the following example, the first ACL grants read and describe permissions for a specific topic named **my-topic**. The **resource.patternType** is set to **literal**, which means that the resource name must match exactly.

The second ACL grants read permissions for a specific consumer group named **my-group**. The **resource.patternType** is set to **prefix**, which means that the resource name must match the prefix.

Example ACL configuration in the KafkaUser resource

```
apiVersion: {KafkaUserApiVersion}
kind: KafkaUser
metadata:
  name: my-user
  labels:
    strimzi.io/cluster: my-cluster
spec:
  # ...
  authorization:
    type: simple
    acls:
      - resource:
          type: topic
          name: my-topic
          patternType: literal
        operations:
          - Read
          - Describe
      - resource:
          type: group
          name: my-group
          patternType: prefix
        operations:
          - Read
```



NOTE

If you specify **tls-external** as an authentication option when configuring the Kafka user, you can use your own client certificates rather than those generated by the User Operator.

5.2. SETTING UP CLIENTS FOR SECURE ACCESS

After you have set up listeners on your Kafka brokers to support secure connections, the next step is to configure your client applications to use these listeners to communicate with the Kafka cluster. This involves providing the appropriate security settings for each client to authenticate with the cluster based on the security mechanisms configured on the listener.

5.2.1. Configuring security protocols

Configure the security protocol used by your client application to match the protocol configured on a Kafka broker listener. For example, use **SSL** (Secure Sockets Layer) for TLS authentication or **SASL_SSL** for SASL (Simple Authentication and Security Layer over SSL) authentication with TLS encryption. Add a truststore and keystore to your client configuration that supports the authentication mechanism required to access the Kafka cluster.

Truststore

The truststore contains the public certificates of the trusted certificate authority (CA) that are used to verify the authenticity of a Kafka broker. When the client connects to a secure Kafka broker, it might need to verify the identity of the broker.

Keystore

The keystore contains the client's private key and its public certificate. When the client wants to authenticate itself to the broker, it presents its own certificate.

If you are using TLS authentication, your Kafka client configuration requires a truststore and keystore to connect to a Kafka cluster. If you are using SASL SCRAM-SHA-512, authentication is performed through the exchange of username and password credentials, rather than digital certificates, so a keystore is not required. SCRAM-SHA-512 is a more lightweight mechanism, but it is not as secure as using certificate-based authentication.

NOTE

If you have your own certificate infrastructure in place and use certificates from a third-party CA, then the client's default truststore will likely already contain the public CA certificates and you do not need to add them to the client's truststore. The client automatically trusts the server's certificate if it is signed by one of the public CA certificates that is already included in the default truststore.

You can create a **config.properties** file to specify the authentication credentials used by the client application.

In the following example, the **security.protocol** is set to **SSL** to enable TLS authentication and encryption between the client and broker.

The **ssl.truststore.location** and **ssl.truststore.password** properties specify the location and password of the truststore. The **ssl.keystore.location** and **ssl.keystore.password** properties specify the location and password of the keystore.

The PKCS #12 (Public-Key Cryptography Standards #12) file format is used. You can also use the base64-encoded PEM (Privacy Enhanced Mail) format.

Example client configuration properties for TLS authentication

```
bootstrap.servers = my-cluster-kafka-bootstrap:9093
security.protocol = SSL
ssl.truststore.location = /path/to/ca.p12
ssl.truststore.password = truststore-password
ssl.keystore.location = /path/to/user.p12
ssl.keystore.password = keystore-password
client.id = my-client
```

In the following example, the **security.protocol** is set to **SASL_SSL** to enable SASL authentication with TLS encryption between the client and broker. If you only need authentication and not encryption, you can use the **SASL** protocol. The specified SASL mechanism for authentication is **SCRAM-SHA-512**. Different authentication mechanisms can be used. **sasl.jaas.config** properties specify the authentication credentials.

Example client configuration properties for SCRAM-SHA-512 authentication

```
bootstrap.servers = my-cluster-kafka-bootstrap:9093
security.protocol = SASL_SSL
sasl.mechanism = SCRAM-SHA-512
sasl.jaas.config = org.apache.kafka.common.security.scram.ScramLoginModule required \
  username = "user" \
  password = "secret";
ssl.truststore.location = path/to/truststore.p12
ssl.truststore.password = truststore_password
ssl.truststore.type = PKCS12
client.id = my-client
```

NOTE

For applications that do not support PEM format, you can use a tool like OpenSSL to convert PEM files to PKCS #12 format.

5.2.2. Configuring permitted TLS versions and cipher suites

You can incorporate SSL configuration and cipher suites to further secure TLS-based communication between your client application and a Kafka cluster. Specify the supported TLS versions and cipher suites in the configuration for the Kafka broker. You can also add the configuration to your clients if you wish to limit the TLS versions and cipher suites they use. The configuration on the client should only use protocols and cipher suites that are enabled on the brokers.

In the following example, SSL is enabled using **security.protocol** for communication between Kafka brokers and client applications. You specify cipher suites as a comma-separated list. The **ssl.cipher.suites property** is a comma-separated list of cipher suites that the client is allowed to use.

Example SSL configuration properties for Kafka brokers

```
security.protocol: "SSL"
ssl.enabled.protocols: "TLSv1.3", "TLSv1.2"
ssl.protocol: "TLSv1.3"
```

```
ssl.cipher.suites: "TLS_AES_256_GCM_SHA384"
```

The **ssl.enabled.protocols** property specifies the available TLS versions that can be used for secure communication between the cluster and its clients. In this case, both **TLSv1.3** and **TLSv1.2** are enabled. The **ssl.protocol** property sets the default TLS version for all connections, and it must be chosen from the enabled protocols. By default, clients communicate using **TLSv1.3**. If a client only supports TLSv1.2, it can still connect to the broker and communicate using that supported version. Similarly, if the configuration is on the client and the broker only supports TLSv1.2, the client uses the supported version.

The cipher suites supported by Apache Kafka depend on the version of Kafka you are using and the underlying environment. Check for the latest supported cipher suites that provide the highest level of security.

5.2.3. Using Access Control Lists (ACLs)

You do not have to configure anything explicitly for ACLS in your client application. The ACLs are enforced on the server side by the Kafka broker. When the client sends a request to the server to produce or consume data, the server checks the ACLs to determine if the client (user) is authorized to perform the requested operation. If the client is authorized, the request is processed; otherwise, the request is denied and an error is returned. However, the client must still be authenticated and using the appropriate security protocol to enable a secure connection with the Kafka cluster.

If you are using Access Control Lists (ACLs) on your Kafka brokers, make sure that ACLs are properly set up to restrict client access to the topics and operations that you want to control. If you are using Open Policy Agent (OPA) policies to manage access, authorization rules are configured in the policies, so you won't need specify ACLs against the Kafka brokers. OAuth 2.0 gives some flexibility: you can use the OAuth 2.0 provider to manage ACLs; or use OAuth 2.0 and Kafka's **simple** authorization to manage the ACLs.

NOTE

ACLs apply to most types of requests and are not limited to produce and consume operations. For example, ACLS can be applied to read operations like describing topics or write operations like creating new topics.

5.2.4. Using OAuth 2.0 for token-based access

Use the OAuth 2.0 open standard for authorization with AMQ Streams to enforce authorization controls through an OAuth 2.0 provider. OAuth 2.0 provides a secure way for applications to access user data stored in other systems. An authorization server can issue access tokens to client applications that grant access to a Kafka cluster.

The following steps describe the general approach to set up and use OAuth 2.0 for token validation:

1. Configure the authorization server with broker and client credentials, such as a client ID and secret.
2. Obtain the OAuth 2.0 credentials from the authorization server.
3. Configure listeners on the Kafka brokers with OAuth 2.0 credentials and to interact with the authorization server.
4. Add the Oauth 2.0 dependency to the client library.

5. Configure your Kafka client with OAuth 2.0 credentials and to interact with the authorization server..

6. Obtain an access token at runtime, which authenticates the client with the OAuth 2.0 provider.

If you have a listener configured for OAuth 2.0 on your Kafka broker, you can set up your client application to use OAuth 2.0. In addition to the standard Kafka client configurations to access the Kafka cluster, you must include specific configurations for OAuth 2.0 authentication. You must also make sure that the authorization server you are using is accessible by the Kafka cluster and client application.

Specify a SASL (Simple Authentication and Security Layer) security protocol and mechanism. In a production environment, the following settings are recommended:

- The **SASL_SSL** protocol for TLS encrypted connections.
- The **OAUTHBEARER** mechanism for credentials exchange using a bearer token

A JAAS (Java Authentication and Authorization Service) module implements the SASL mechanism. The configuration for the mechanism depends on the authentication method you are using. For example, using credentials exchange you add an OAuth 2.0 access token endpoint, access token, client ID, and client secret. A client connects to the token endpoint (URL) of the authorization server to check if a token is still valid. You also need a truststore that contains the public key certificate of the authorization server for authenticated access.

Example client configuration properties for OAauth 2.0

```
bootstrap.servers = my-cluster-kafka-bootstrap:9093
security.protocol = SASL_SSL
sasl.mechanism = OAUTHBEARER
# ...
sasl.jaas.config = org.apache.kafka.common.security.oauthbearer.OAuthBearerLoginModule required
\
    oauth.token.endpoint.uri = "https://localhost:9443/oauth2/token" \
    oauth.access.token = <access_token> \
    oauth.client.id = "<client_id>" \
    oauth.client.secret = "<client_secret>" \
    oauth.ssl.truststore.location = "/<truststore_location>/oauth-truststore.p12" \
    oauth.ssl.truststore.password = "<truststore_password>" \
    oauth.ssl.truststore.type = "PKCS12" \
```

For more information on setting up your brokers to use OAuth 2.0, see the following guides:

- Deploying and Upgrading AMQ Streams on OpenShift
- Using AMQ Streams on RHEL

5.2.5. Using Open Policy Agent (OPA) access policies

Use the Open Policy Agent (OPA) policy agent with AMQ Streams to evaluate requests to connect to your Kafka cluster against access policies. Open Policy Agent (OPA) is a policy engine that manages authorization policies. Policies centralize access control, and can be updated dynamically, without requiring changes to the client application. For example, you can create a policy that allows only certain users (clients) to produce and consume messages to a specific topic.

AMQ Streams uses the Open Policy Agent plugin for Kafka authorization as the authorizer.

The following steps describe the general approach to set up and use OPA:

1. Set up an instance of the OPA server.
2. Define policies that provide the authorization rules that govern access to the Kafka cluster.
3. Create configuration for the Kafka brokers to accept OPA authorization and interact with the OPA server.
4. Configure your Kafka client to provide the credentials for authorized access to the Kafka cluster.

If you have a listener configured for OPA on your Kafka broker, you can set up your client application to use OPA. In the listener configuration, you specify a URL to connect to the OPA server and authorize your client application. In addition to the standard Kafka client configurations to access the Kafka cluster, you must add the credentials to authenticate with the Kafka broker. The broker checks if the client has the necessary authorization to perform a requested operation, by sending a request to the OPA server to evaluate the authorization policy. You don't need a truststore or keystore to secure communication as the policy engine enforces authorization policies.

Example client configuration properties for OPA authorization

```
bootstrap.servers = my-cluster-kafka-bootstrap:9093
security.protocol = SASL_SSL
sasl.mechanism = SCRAM-SHA-512
sasl.jaas.config = org.apache.kafka.common.security.scram.ScramLoginModule required \
  username = "user" \
  password = "secret";
# ...
```

NOTE

Red Hat does not support the OPA server.

For more information on setting up your brokers to use OPA, see the following guides:

- Deploying and Upgrading AMQ Streams on OpenShift
- Using AMQ Streams on RHEL

5.2.6. Using transactions when streaming messages

By configuring transaction properties in your brokers and producer client application, you can ensure that messages are processed in a single transaction. Transactions add reliability and consistency to the streaming of messages.

Transactions are always enabled on brokers. You can change the default configuration using the following properties:

Example Kafka broker configuration properties for transactions

```
transaction.state.log.replication.factor = 3
transaction.state.log.min.isr = 2
transaction.abort.timed.out.transaction.cleanup.interval.ms = 3600000
```

This is a typical configuration for a production environment, which creates 3 replicas for the internal **__transaction_state** topic. The **__transaction_state** topic stores information about the transactions in progress. A minimum of 2 in-sync replicas are required for the transaction logs. The cleanup interval is the time between checks for timed-out transactions and a clean up the corresponding transaction logs.

To add transaction properties to a client configuration, you set the following properties for producers and consumers.

Example producer client configuration properties for transactions

```
transactional.id = unique-transactional-id
enable.idempotence = true
max.in.flight.requests.per.connection = 5
acks = all
retries=2147483647
transaction.timeout.ms = 30000
delivery.timeout = 25000
```

The transactional ID allows the Kafka broker to keep track of the transactions. It is a unique identifier for the producer and should be used with a specific set of partitions. If you need to perform transactions for multiple sets of partitions, you need to use a different transactional ID for each set. Idempotence is enabled to avoid the producer instance creating duplicate messages. With idempotence, messages are tracked using a producer ID and sequence number. When the broker receives the message, it checks the producer ID and sequence number. If a message with the same producer ID and sequence number has already been received, the broker discards the duplicate message.

The maximum number of in-flight requests is set to 5 so that transactions are processed in the order they are sent. A partition can have up to 5 in-flight requests without compromising the ordering of messages.

By setting **acks** to **all**, the producer waits for acknowledgments from all in-sync replicas of the topic partitions to which it is writing before considering the transaction as complete. This ensures that the messages are durably written (committed) to the Kafka cluster, and that they will not be lost even in the event of a broker failure.

The transaction timeout specifies the maximum amount of time the client has to complete a transaction before it times out. The delivery timeout specifies the maximum amount of time the producer waits for a broker acknowledgement of message delivery before it times out. To ensure that messages are delivered within the transaction period, set the delivery timeout to be less than the transaction timeout. Consider network latency and message throughput, and allow for temporary failures, when specifying **retries** for the number of attempts to resend a failed message request.

Example consumer client configuration properties for transactions

```
group.id = my-group-id
isolation.level = read_committed
enable.auto.commit = false
```

The **read_committed** isolation level specifies that the consumer only reads messages for a transaction that has completed successfully. The consumer does not process any messages that are part of an ongoing or failed transaction. This ensures that the consumer only reads messages that are part of a fully complete transaction.

When using transactions to stream messages, it is important to set **enable.auto.commit** to **false**. If set to **true**, the consumer periodically commits offsets without consideration to transactions. This means

that the consumer may commit messages before a transaction has fully completed. By setting **enable.auto.commit** to **false**, the consumer only reads and commits messages that have been fully written and committed to the topic as part of a transaction.

CHAPTER 6. DEVELOPING A KAFKA CLIENT

Create a Kafka client in your preferred programming language and connect it to AMQ Streams.

To interact with a Kafka cluster, client applications need to be able to produce and consume messages. To develop and configure a basic Kafka client application, as a minimum, you must do the following:

- Set up configuration to connect to a Kafka cluster
- Use producers and consumers to send and receive messages

Setting up the basic configuration for connecting to a Kafka cluster and using producers and consumers is the first step in developing a Kafka client. After that, you can expand into improving the inputs, security, performance, error handling, and functionality of the client application.

Prerequisites

You can create a client properties file that contains property values for the following:

- Basic configuration to connect to the Kafka cluster
- Configuration for securing the connection

Procedure

1. Choose a Kafka client library for your programming language, e.g. Java, Python, .NET, etc. Only client libraries built by Red Hat are supported for AMQ Streams. Currently, AMQ Streams only provides a Java client library.
2. Install the library, either through a package manager or manually by downloading the library from its source.
3. Import the necessary classes and dependencies for your Kafka client in your code.
4. Create a Kafka consumer or producer object, depending on the type of client you want to create.
   A client can be a Kafka consumer, producer, Streams processor, and admin.
5. Provide the configuration properties to connect to the Kafka cluster, including the broker address, port, and credentials if necessary.
   For a local Kafka deployment, you might start with an address like **localhost:9092**. However, when working with a Kafka cluster managed by AMQ Streams, you can obtain the bootstrap address from the **Kafka** custom resource status using an **oc** command:

   ```
   oc get kafka <kafka_cluster_name> -o=jsonpath='{.status.listeners[*].bootstrapServers}{"\n"}'
   ```

   This command retrieves the bootstrap addresses exposed by listeners for client connections on a Kafka cluster.
6. Use the Kafka consumer or producer object to subscribe to topics, produce messages, or retrieve messages from the Kafka cluster.
7. Pay attention to error handling; it's vitally important when connecting and communicating with Kafka, especially in production systems where high availability and ease of operations are valued. Effective error handling is a key differentiator between a prototype and a production-grade application, and it applies not only to Kafka but also to any robust software system.

6.1. EXAMPLE KAFKA PRODUCER APPLICATION

This Java-based Kafka producer application is an example of a self-contained application that produces messages to a Kafka topic. The client uses the Kafka **Producer** API to send messages asynchronously, with some error handling.

The client implements the **Callback** interface for message handling.

To run the Kafka producer application, you execute the **main** method in the **Producer** class. The client generates a random byte array as the message payload using the **randomBytes** method. The client produces messages to a specified Kafka topic until **NUM_MESSAGES** messages (50 in the example configuration) have been sent. The producer is thread-safe, allowing multiple threads to use a single producer instance.

Kafka producer instances are designed to be thread-safe, allowing multiple threads to share a single producer instance.

This example client provides a basic foundation for building more complex Kafka producers for specific use cases. You can incorporate additional functionality, such as implementing secure connections.

Prerequisites

- Kafka brokers running on the specified **BOOTSTRAP_SERVERS**
- A Kafka topic named **TOPIC_NAME** to which messages are produced.
- Client dependencies

Before implementing the Kafka producer application, your project must include the necessary dependencies. For a Java-based Kafka client, include the Kafka client JAR. This JAR file contains the Kafka libraries required for building and running the client.

For information on how to add the dependencies to a **pom.xml** file in a Maven project, see Section 3.1, "Adding a Kafka clients dependency to your Maven project".

Configuration

You can configure the producer application through the following constants specified in the **Producer** class:

BOOTSTRAP_SERVERS
: The address and port to connect to the Kafka brokers.

TOPIC_NAME
: The name of the Kafka topic to produce messages to.

NUM_MESSAGES
: The number of messages to produce before stopping.

MESSAGE_SIZE_BYTES
: The size of each message in bytes.

PROCESSING_DELAY_MS
: The delay in milliseconds between sending messages. This can simulate message processing time, which is useful for testing.

Example producer application

```
import java.util.Properties;
import java.util.Random;
import java.util.UUID;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.atomic.AtomicLong;

import org.apache.kafka.clients.producer.Callback;
import org.apache.kafka.clients.producer.KafkaProducer;
import org.apache.kafka.clients.producer.ProducerConfig;
import org.apache.kafka.clients.producer.ProducerRecord;
import org.apache.kafka.clients.producer.RecordMetadata;
import org.apache.kafka.common.errors.RetriableException;
import org.apache.kafka.common.serialization.ByteArraySerializer;
import org.apache.kafka.common.serialization.LongSerializer;

public class Producer implements Callback {
    private static final Random RND = new Random(0);
    private static final String BOOTSTRAP_SERVERS = "localhost:9092";
    private static final String TOPIC_NAME = "my-topic";
    private static final long NUM_MESSAGES = 50;
    private static final int MESSAGE_SIZE_BYTES = 100;
    private static final long PROCESSING_DELAY_MS = 1000L;

    protected AtomicLong messageCount = new AtomicLong(0);

    public static void main(String[] args) {
        new Producer().run();
    }

    public void run() {
        System.out.println("Running producer");
        try (var producer = createKafkaProducer()) { 1
            byte[] value = randomBytes(MESSAGE_SIZE_BYTES); 2
            while (messageCount.get() < NUM_MESSAGES) { 3
                sleep(PROCESSING_DELAY_MS); 4
                producer.send(new ProducerRecord<>(TOPIC_NAME, messageCount.get(), value), this);
5
                messageCount.incrementAndGet();
            }
        }
    }

    private KafkaProducer<Long, byte[]> createKafkaProducer() {
        Properties props = new Properties(); 6
        props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, BOOTSTRAP_SERVERS); 7
        props.put(ProducerConfig.CLIENT_ID_CONFIG, "client-" + UUID.randomUUID()); 8
        props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, LongSerializer.class); 9
        props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, ByteArraySerializer.class);
        return new KafkaProducer<>(props);
    }

    private void sleep(long ms) { 10
        try {
            TimeUnit.MILLISECONDS.sleep(ms);
        } catch (InterruptedException e) {
```

```
            throw new RuntimeException(e);
        }
    }

    private byte[] randomBytes(int size) { 11
        if (size <= 0) {
            throw new IllegalArgumentException("Record size must be greater than zero");
        }
        byte[] payload = new byte[size];
        for (int i = 0; i < payload.length; ++i) {
            payload[i] = (byte) (RND.nextInt(26) + 65);
        }
        return payload;
    }

    private boolean retriable(Exception e) { 12
        if (e instanceof IllegalArgumentException
            || e instanceof UnsupportedOperationException
            || !(e instanceof RetriableException)) {
            return false;
        } else {
            return true;
        }
    }

    @Override
    public void onCompletion(RecordMetadata metadata, Exception e) { 13
        if (e != null) {
            System.err.println(e.getMessage());
            if (!retriable(e)) {
                e.printStackTrace();
                System.exit(1);
            }
        } else {
            System.out.printf("Record sent to %s-%d with offset %d%n",
                metadata.topic(), metadata.partition(), metadata.offset());
        }
    }
}
```

1. The client creates a Kafka producer using the **createKafkaProducer** method. The producer sends messages to the Kafka topic asynchronously.
2. A byte array is used as the payload for each message sent to the Kafka topic.
3. The maximum number of messages sent is determined by the **NUM_MESSAGES** constant value.
4. The message rate is controlled with a delay between each message sent.
5. The producer passes the topic name, the message count value, and the message value.
6. The client creates the **KafkaProducer** instance using the provided configuration. You can use a properties file or add the configuration directly. For more information on the basic configuration, see Chapter 4, *Configuring client applications for connecting to a Kafka cluster* .
7. The connection to the Kafka brokers.

8. A unique client ID for the producer using a randomly generated UUID. A client ID is not required, but it is useful to track the source of requests.

9. The appropriate serializer classes for handling keys and values as byte arrays.

10. Method to introduce a delay to the message sending process for a specified number of milliseconds. If the thread responsible for sending messages is interrupted while paused, it throws an **InterruptedException** error.

11. Method to create a random byte array of a specific size, which serves as the payload for each message sent to the Kafka topic. The method generates a random integer and adds **65** to represent an uppercase letter in ascii code (65 is **A**, 66 is **B**, and so on). The ascii code is stored as a single byte in the payload array. If the payload size is not greater than zero, it throws an **IllegalArgumentException**.

12. Method to check whether to retry sending a message following an exception. The Kafka producer automatically handles retries for certain errors, such as connection errors. You can customize this method to include other errors. Returns **false** for null and specified exceptions, or those that do not implement the **RetriableException** interface.

13. Method called when a message has been acknowledged by the Kafka broker. On success, a message is printed with the details of the topic, partition, and offset position for the message. If an error ocurred when sending the message, an error message is printed. The method checks the exception and takes appropriate action based on whether it's a fatal or non-fatal error. If the error is non-fatal, the message sending process continues. If the error is fatal, a stack trace is printed and the producer is terminated.

Error handling

Fatal exceptions caught by the producer application:

InterruptedException

Error thrown when the current thread is interrupted while paused. Interruption typically occurs when stopping or shutting down the producer. The exception is rethrown as a **RuntimeException**, which terminates the producer.

IllegalArgumentException

Error thrown when the producer receives invalid or inappropriate arguments. For example, the exception is thrown if the topic is missing.

UnsupportedOperationException

Error thrown when an operation is not supported or a method is not implemented. For example, the exception is thrown if an attempt is made to use an unsupported producer configuration or call a method that is not supported by the **KafkaProducer** class.

Non-fatal exceptions caught by the producer application:

RetriableException

Error thrown for any exception that implements the **RetriableException** interface provided by the Kafka client library.

With non-fatal errors, the producer continues to send messages.

NOTE

By default, Kafka operates with at-least-once message delivery semantics, which means that messages can be delivered more than once in certain scenarios, potentially leading to duplicates. To avoid this risk, consider enabling transactions in your Kafka producer . Transactions provide stronger guarantees of exactly-once delivery. Additionally, you can use the **retries** configuration property to control how many times the producer will retry sending a message before giving up. This setting affects how many times the **retriable** method may return **true** during a message send error.

6.2. EXAMPLE KAFKA CONSUMER APPLICATION

This Java-based Kafka consumer application is an example of a self-contained application that consumes messages from a Kafka topic. The client uses the Kafka **Consumer** API to fetch and process messages from a specified topic asynchronously, with some error handling. It follows at-least-once semantics by committing offsets after successfully processing messages.

The client implements the **ConsumerRebalanceListener** interface for partition handling and the **OffsetCommitCallback** interface for committing offsets.

To run the Kafka consumer application, you execute the **main** method in the **Consumer** class. The client consumes messages from the Kafka topic until **NUM_MESSAGES** messages (50 in the example configuration) have been consumed. The consumer is not designed to be safely accessed concurrently by multiple threads.

This example client provides a basic foundation for building more complex Kafka consumers for specific use cases. You can incorporate additional functionality, such as implementing secure connections.

Prerequisites

- Kafka brokers running on the specified **BOOTSTRAP_SERVERS**
- A Kafka topic named **TOPIC_NAME** from which messages are consumed.
- Client dependencies

Before implementing the Kafka consumer application, your project must include the necessary dependencies. For a Java-based Kafka client, include the Kafka client JAR. This JAR file contains the Kafka libraries required for building and running the client.

For information on how to add the dependencies to a **pom.xml** file in a Maven project, see Section 3.1, “Adding a Kafka clients dependency to your Maven project”.

Configuration

You can configure the consumer application through the following constants specified in the **Consumer** class:

BOOTSTRAP_SERVERS

The address and port to connect to the Kafka brokers.

GROUP_ID

The consumer group identifier.

POLL_TIMEOUT_MS

The maximum time to wait for new messages during each poll.

TOPIC_NAME

The name of the Kafka topic to consume messages from.

NUM_MESSAGES

The number of messages to consume before stopping.

PROCESSING_DELAY_MS

The delay in milliseconds between sending messages. This can simulate message processing time, which is useful for testing.

Example consumer application

```
import java.util.Collection;
import java.util.HashMap;
import java.util.Map;
import java.util.Properties;
import java.util.UUID;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.atomic.AtomicLong;

import org.apache.kafka.clients.consumer.ConsumerConfig;
import org.apache.kafka.clients.consumer.ConsumerRebalanceListener;
import org.apache.kafka.clients.consumer.ConsumerRecord;
import org.apache.kafka.clients.consumer.ConsumerRecords;
import org.apache.kafka.clients.consumer.KafkaConsumer;
import org.apache.kafka.clients.consumer.NoOffsetForPartitionException;
import org.apache.kafka.clients.consumer.OffsetAndMetadata;
import org.apache.kafka.clients.consumer.OffsetCommitCallback;
import org.apache.kafka.clients.consumer.OffsetOutOfRangeException;
import org.apache.kafka.common.TopicPartition;
import org.apache.kafka.common.errors.RebalanceInProgressException;
import org.apache.kafka.common.errors.RetriableException;
import org.apache.kafka.common.serialization.ByteArrayDeserializer;
import org.apache.kafka.common.serialization.LongDeserializer;

import static java.time.Duration.ofMillis;
import static java.util.Collections.singleton;

public class Consumer implements ConsumerRebalanceListener, OffsetCommitCallback {
    private static final String BOOTSTRAP_SERVERS = "localhost:9092";
    private static final String GROUP_ID = "my-group";
    private static final long POLL_TIMEOUT_MS = 1_000L;
    private static final String TOPIC_NAME = "my-topic";
    private static final long NUM_MESSAGES = 50;
    private static final long PROCESSING_DELAY_MS = 1_000L;

    private KafkaConsumer<Long, byte[]> kafkaConsumer;
    protected AtomicLong messageCount = new AtomicLong(0);
    private Map<TopicPartition, OffsetAndMetadata> pendingOffsets = new HashMap<>();

    public static void main(String[] args) {
        new Consumer().run();
    }

    public void run() {
        System.out.println("Running consumer");
```

```
        try (var consumer = createKafkaConsumer()) { (1)
            kafkaConsumer = consumer;
            consumer.subscribe(singleton(TOPIC_NAME), this); (2)
            System.out.printf("Subscribed to %s%n", TOPIC_NAME);
            while (messageCount.get() < NUM_MESSAGES) { (3)
                try {
                    ConsumerRecords<Long, byte[]> records =
consumer.poll(ofMillis(POLL_TIMEOUT_MS)); (4)
                    if (!records.isEmpty()) { (5)
                        for (ConsumerRecord<Long, byte[]> record : records) {
                            System.out.printf("Record fetched from %s-%d with offset %d%n",
                                record.topic(), record.partition(), record.offset());
                            sleep(PROCESSING_DELAY_MS); (6)

                            pendingOffsets.put(new TopicPartition(record.topic(), record.partition()), (7)
                                new OffsetAndMetadata(record.offset() + 1, null));
                            if (messageCount.incrementAndGet() == NUM_MESSAGES) {
                                break;
                            }
                        }
                        consumer.commitAsync(pendingOffsets, this); (8)
                        pendingOffsets.clear();
                    }
                } catch (OffsetOutOfRangeException | NoOffsetForPartitionException e) { (9)
                    System.out.println("Invalid or no offset found, and auto.reset.policy unset, using latest");
                    consumer.seekToEnd(e.partitions());
                    consumer.commitSync();
                } catch (Exception e) {
                    System.err.println(e.getMessage());
                    if (!retriable(e)) {
                        e.printStackTrace();
                        System.exit(1);
                    }
                }
            }
        }
    }

    private KafkaConsumer<Long, byte[]> createKafkaConsumer() {
        Properties props = new Properties(); (10)
        props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, BOOTSTRAP_SERVERS); (11)
        props.put(ConsumerConfig.CLIENT_ID_CONFIG, "client-" + UUID.randomUUID()); (12)
        props.put(ConsumerConfig.GROUP_ID_CONFIG, GROUP_ID); (13)
        props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, LongDeserializer.class);
(14)
        props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
ByteArrayDeserializer.class);
        props.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, false); (15)
        props.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest"); (16)
        return new KafkaConsumer<>(props);
    }

    private void sleep(long ms) { (17)
        try {
```

```
            TimeUnit.MILLISECONDS.sleep(ms);
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
    }

    private boolean retriable(Exception e) { 18
        if (e == null) {
            return false;
        } else if (e instanceof IllegalArgumentException
            || e instanceof UnsupportedOperationException
            || !(e instanceof RebalanceInProgressException)
            || !(e instanceof RetriableException)) {
            return false;
        } else {
            return true;
        }
    }

    @Override
    public void onPartitionsAssigned(Collection<TopicPartition> partitions) { 19
        System.out.printf("Assigned partitions: %s%n", partitions);
    }

    @Override
    public void onPartitionsRevoked(Collection<TopicPartition> partitions) { 20
        System.out.printf("Revoked partitions: %s%n", partitions);
        kafkaConsumer.commitSync(pendingOffsets);
        pendingOffsets.clear();
    }

    @Override
    public void onPartitionsLost(Collection<TopicPartition> partitions) { 21
        System.out.printf("Lost partitions: {}", partitions);
    }

    @Override
    public void onComplete(Map<TopicPartition, OffsetAndMetadata> map, Exception e) { 22
        if (e != null) {
            System.err.println("Failed to commit offsets");
            if (!retriable(e)) {
                e.printStackTrace();
                System.exit(1);
            }
        }
    }
}
```

1. The client creates a Kafka consumer using the **createKafkaConsumer** method.
2. The consumer subscribes to a specific topic. After subscribing to the topic, a confirmation message is printed.
3. The maximum number of messages consumed is determined by the **NUM_MESSAGES** constant value.

4. The next poll to fetch messages must be called within **session.timeout.ms** to avoid a rebalance.

5. A condition to check that the **records** object containing the batch messages fetched from Kafka is not empty. If the **records** object is empty, there are no new messages to process and the process is skipped.

6. Method to introduce a delay to the message fetching process for a specified number of milliseconds.

7. The consumer uses a **pendingOffsets** map to store the offsets of the consumed messages that need to be committed.

8. After processing a batch of messages, the consumer asynchronously commits the offsets using the **commitAsync** method, implementing at-least-once semantics.

9. A catch to handle non-fatal and fatal errors when consuming messages and auto-reset policy is not set. For non-fatal errors, the consumer seeks to the end of the partition and starts consuming from the latest available offset. If an exception cannot be retried, a stack trace is printed, and the consumer is terminated.

10. The client creates the **KafkaConsumer** instance using the provided configuration. You can use a properties file or add the configuration directly. For more information on the basic configuration, see Chapter 4, *Configuring client applications for connecting to a Kafka cluster* .

11. The connection to the Kafka brokers.

12. A unique client ID for the producer using a randomly generated UUID. A client ID is not required, but it is useful to track the source of requests.

13. The group ID for consumer coordination of assignments to partitions.

14. The appropriate deserializer classes for handling keys and values as byte arrays.

15. Configuration to disable automatic offset commits.

16. Configuration for the consumer to start consuming messages from the earliest available offset when no committed offset is found for a partition.

17. Method to introduce a delay to the message consuming process for a specified number of milliseconds. If the thread responsible for sending messages is interrupted while paused, it throws an **InterruptedException** error.

18. Method to check whether to retry committing a message following an exception. Null and specified exceptions are not retried, nor are exceptions that do not implement the **RebalanceInProgressException** or **RetriableException** interfaces. You can customize this method to include other errors.

19. Method to print a message to the console indicating the list of partitions that have been assigned to the consumer.

20. Method called when the consumer is about to lose ownership of partitions during a consumer group rebalance. The method prints the list of partitions that are being revoked from the consumer. Any pending offsets are committed.

21. Method called when the consumer loses ownership of partitions during a consumer group rebalance, but failed to commit any pending offsets. The method prints the list of partitions lost by the consumer.

22 Method called when the consumer is committing offsets to Kafka. If an error ocurred when committing an offset, an error message is printed. The method checks the exception and takes

Error handling

Fatal exceptions caught by the consumer application:

InterruptedException

Error thrown when the current thread is interrupted while paused. Interruption typically occurs when stopping or shutting down the consumer. The exception is rethrown as a **RuntimeException**, which terminates the consumer.

IllegalArgumentException

Error thrown when the consumer receives invalid or inappropriate arguments. For example, the exception is thrown if the topic is missing.

UnsupportedOperationException

Error thrown when an operation is not supported or a method is not implemented. For example, the exception is thrown if an attempt is made to use an unsupported consumer configuration or call a method that is not supported by the **KafkaConsumer** class.

Non-fatal exceptions caught by the consumer application:

OffsetOutOfRangeException

Error thrown when the consumer attempts to seek to an invalid offset for a partition, typically when the offset is outside the valid range of offsets for that partition, and auto-reset policy is not enabled. To recover, the consumer seeks to the end of the partition to commit the offset synchronously (**commitSync**). If auto-reset policy is enabled, the consumer seeks to the start or end of the partition depending on the setting.

NoOffsetForPartitionException

Error thrown when there is no committed offset for a partition or the requested offset is invalid, and auto-reset policy is not enabled. To recover, the consumer seeks to the end of the partition to commit the offset synchronously (**commitSync**). If auto-reset policy is enabled, the consumer seeks to the start or end of the partition depending on the setting.

RebalanceInProgressException

Error thrown during a consumer group rebalance when partitions are being assigned. Offset commits cannot be completed when the consumer is undergoing a rebalance.

RetriableException

Error thrown for any exception that implements the **RetriableException** interface provided by the Kafka client library.

With non-fatal errors, the consumer continues to process messages.

6.3. USING COOPERATIVE REBALANCING WITH CONSUMERS

Kafka consumers use a partition assignment strategy determined by the rebalancing protocol in place. By default, Kafka employs the **RangeAssignor** protocol, which involves consumers relinquishing their partition assignments during a rebalance, leading to potential service disruptions.

To improve efficiency and reduce downtime, you can switch to the **CooperativeStickyAssignor** protocol, a cooperative rebalancing approach. Unlike the default protocol, cooperative rebalancing enables consumers to work together, retaining their partition assignments during a rebalance, and releasing partitions only when necessary to achieve a balance within the consumer group.

Procedure

1. In the consumer configuration, use the **partition.assignment.strategy** property to switch to using **CooperativeStickyAssignor** as the protocol. For example, if the current configuration is **partition.assignment.strategy=RangeAssignor, CooperativeStickyAssignor**, update it to **partition.assignment.strategy=CooperativeStickyAssignor**.
   Instead of modifying the consumer configuration file directly, you can also set the partition assignment strategy using **props.put** in the consumer application code:

   ```
   # ...
   props.put(ConsumerConfig.PARTITION_ASSIGNMENT_STRATEGY_CONFIG,
   "org.apache.kafka.clients.consumer.CooperativeStickyAssignor");
   # ...
   ```

2. Restart each consumer in the group one at a time, allowing them to rejoin the group after each restart.

WARNING

After switching to the **CooperativeStickyAssignor** protocol, a **RebalanceInProgressException** may occur during consumer rebalancing, leading to unexpected stoppages of multiple Kafka clients in the same consumer group. Additionally, this issue may result in the duplication of uncommitted messages, even if Kafka consumers have not changed their partition assignments during rebalancing. If you are using automatic offset commits (**enable.auto.commit=true**), you don't need to make any changes. If you are manually committing offsets (**enable.auto.commit=false**), and a **RebalanceInProgressException** occurs during the manual commit, change the consumer implementation to call **poll()** in the next loop to complete the consumer rebalancing process. For more information, see the **CooperativeStickyAssignor** article on the customer portal.

APPENDIX A. USING YOUR SUBSCRIPTION

AMQ Streams is provided through a software subscription. To manage your subscriptions, access your account at the Red Hat Customer Portal.

Accessing Your Account

1. Go to access.redhat.com.
2. If you do not already have an account, create one.
3. Log in to your account.

Activating a Subscription

1. Go to access.redhat.com.
2. Navigate to **My Subscriptions**.
3. Navigate to **Activate a subscription** and enter your 16-digit activation number.

Downloading Zip and Tar Files

To access zip or tar files, use the customer portal to find the relevant files for download. If you are using RPM packages, this step is not required.

1. Open a browser and log in to the Red Hat Customer Portal **Product Downloads** page at access.redhat.com/downloads.
2. Locate the **AMQ Streams for Apache Kafka** entries in the **INTEGRATION AND AUTOMATION** category.
3. Select the desired AMQ Streams product. The **Software Downloads** page opens.
4. Click the **Download** link for your component.

Installing packages with DNF

To install a package and all the package dependencies, use:

```
dnf install <package_name>
```

To install a previously-downloaded package from a local directory, use:

```
dnf install <path_to_download_package>
```

Revised on 2024-04-12 15:50:49 UTC