Distributed Message Service for Kafka

Developer Guide

Issue 03

Date 2023-05-26

HUAWEI

HUAWEI CLOUD COMPUTING TECHNOLOGIES CO., LTD.

Trademarks and Permissions


and other Huawei trademarks are the property of Huawei Technologies Co., Ltd.
All other trademarks and trade names mentioned in this document are the property of their respective holders.

Notice

The purchased products, services and features are stipulated by the contract made between Huawei Cloud and the customer. All or part of the products, services and features described in this document may not be within the purchase scope or the usage scope. Unless otherwise specified in the contract, all statements, information, and recommendations in this document are provided "AS IS" without warranties, guarantees or representations of any kind, either express or implied.

The information in this document is subject to change without notice. Every effort has been made in the preparation of this document to ensure accuracy of the contents, but all statements, information, and recommendations in this document do not constitute a warranty of any kind, express or implied.

Contents

1 Overview

Kafka instances are compatible with Apache Kafka and can be accessed using **open-source Kafka clients**. In addition, if SASL authentication is used, use the DMS certificate.

This document describes how to collect instance connection information, such as the instance connection address and topic name. It also provides examples of accessing an instance in Java, Python, and Go.

The examples only demonstrate how to invoke Kafka APIs for producing and consuming messages. For more information about the APIs provided by Kafka, visit the **Kafka official website**.

Client Network Environment

A client can access a Kafka instance in any of the following modes:

- If the client runs an Elastic Cloud Server (ECS) and is in the same region and VPC as the Kafka instance, the client can access the instance using a private network IP address.
- If the client runs an ECS, and is in the same region but not in the same VPC as the Kafka instance, the client can access the instance using one of the following methods:
  - Establish a VPC peering connection to allow two VPCs to communicate with each other. For details, see section "VPC Peering Connection" in *Virtual Private Cloud User Guide*. Modify the security group of your Kafka instance as required: Allow external access over port 9092 (without SASL) or 9093 (with SASL).
  - Access a Kafka instance on a Kafka client over a private network across VPCs using a VPC endpoint. For details, see section "Accessing Kafka Across VPCs Using VPCEP" in *Distributed Message Service for Kafka User Guide*. Modify the security group to allow port 9011 for external access.
- If the client is not in the same network environment or region as the Kafka instance, the client can access the instance using a public network IP address.

  To use public access, modify the security group of your Kafka instance as required: Allow external access over port 9094 (without SASL) or 9095 (with SASL).

NOTE

The three modes differ only in the connection address for the client to access the instance. This document takes intra-VPC access as an example to describe how to set up the development environment.

If the connection times out or fails, check the network connectivity. You can use telnet to test the connection address and port of the instance.

2 Collecting Connection Information

Before accessing a Kafka instance for message production and consumption, obtain the following information of the instance.

Instance Connection Address and Port

Obtain them from the **Basic Information** page on the Kafka console.

For a cluster Kafka instance that has at least three connection addresses, you are advised to configure all of the connection addresses on the client for high reliability.

For public network access, you can use the public network addresses displayed on the **Basic Information** page.

Figure 2-1 Viewing the connection addresses and ports of brokers of a Kafka instance

Topic name

Obtain the topic name from the **Topics** page of the Kafka instance console.

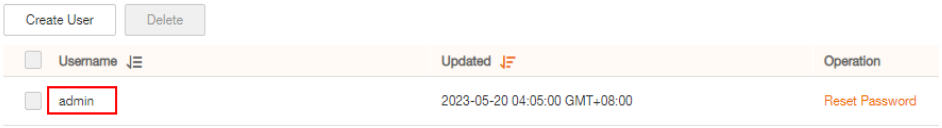
If **Automatic Topic Creation** is disabled, a topic must be created first. Then a Kafka instance can be accessed from a client for message production and consumption.

SASL

If SASL_SSL is enabled for the instance during instance creation, obtain the SASL_SSL username, password, certificate, and mechanism.

- Obtain the username on the **Users** page on the Kafka console. If you forget your password, obtain it again by "resetting SASL_SSL password" in *Distributed Message Service for Kafka User Guide*.

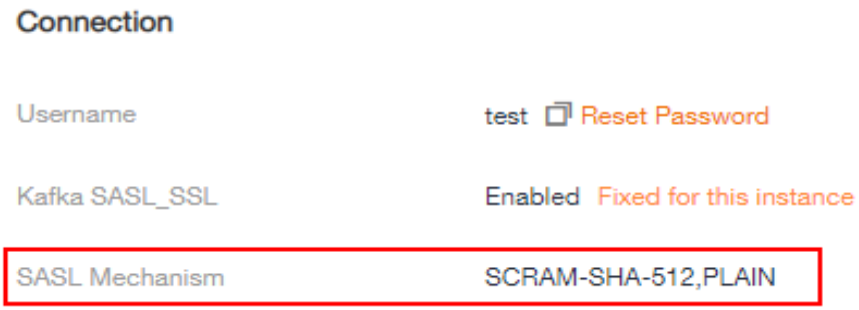
Figure 2-2 Obtaining the SASL_SSL username



- Obtain the SASL mechanism from the **Basic Information** page on the Kafka console.

  If both SCRAM-SHA-512 and PLAIN are enabled, use either of them in connection configurations. For instances that were created much earlier, if **SASL Mechanism** is not displayed on the instance details page, PLAIN is used by default.

  Figure 2-3 SASL mechanism in use

- Obtain the SSL certificate from the **Basic Information** page on the Kafka console.

  JKS certificates are used for accessing instances in Java. CRT certificates are used for accessing instances in Python.

3 Java

3.1 Configuring Kafka Clients in Java

This section describes how to add Kafka clients in Maven, and use the clients to access Kafka instances and produce and consume messages. To check how the demo project runs in IDEA, see **Setting Up the Java Development Environment**.

The Kafka instance connection addresses, topic name, and user information used in the following examples are available in **Collecting Connection Information**.

Adding Kafka Clients in Maven

```
//Kafka instances are based on Kafka 1.1.0/2.3.0/2.7. Use the same version of the client.
    <dependency>
       <groupId>org.apache.kafka</groupId>
       <artifactId>kafka-clients</artifactId>
       <version>1.1.0/2.3.0/2.7.2</version>
    </dependency>
```

Preparing Kafka Configuration Files

The following describes example producer and consumer configuration files. If SASL is not enabled for the Kafka instance, comment out lines regarding the encryption. Otherwise, set configurations for encrypted access.

- Producer configuration file (the **dms.sdk.producer.properties** file in the **message production code**)

  The information in bold is specific to different Kafka instances and must be modified. Other parameters can also be added.

  ```
  #The topic name is in the specific production and consumption code.
  ########################
  #Information about Kafka brokers. ip:port are the connection addresses and ports used by the instance. The values can be obtained by referring to the "Collecting Connection Information" section.
  Example: bootstrap.servers=100.xxx.xxx.87:909x,100.xxx.xxx.69:909x,100.xxx.xxx.155:909x
  bootstrap.servers=ip1:port1,ip2:port2,ip3:port3
  #Producer acknowledgement
  acks=all
  #Method of turning the key into bytes
  key.serializer=org.apache.kafka.common.serialization.StringSerializer
  #Method of turning the value into bytes
  value.serializer=org.apache.kafka.common.serialization.StringSerializer
  ```



```
#Memory available to the producer for buffering
buffer.memory=33554432
#Number of retries
retries=0
########################
#Comment out the following parameters if SASL is not enabled.
########################
# Set the SASL authentication mechanism, username, and password.
#sasl.mechanism is the SASL mechanism. username and password are the SASL username and password. Obtain them by referring to section "Collecting Connection Information". For security purposes, you are advised to encrypt the username and password.
# If the SASL mechanism is PLAIN, the configuration is as follows:
sasl.mechanism=PLAIN
sasl.jaas.config=org.apache.kafka.common.security.plain.PlainLoginModule required \
    username="username" \
    password="password";
# If the SASL mechanism is SCRAM-SHA-512, the configuration is as follows:
sasl.mechanism=SCRAM-SHA-512
sasl.jaas.config=org.apache.kafka.common.security.scram.ScramLoginModule required \
    username="username" \
    password="password";

# Set security.protocol.
# If the security protocol is SASL_SSL, the configuration is as follows:
security.protocol=SASL_SSL
# ssl truststore.location is the path for storing the SSL certificate. The following code uses the path format in Windows as an example. Change the path format based on the actual running environment.
ssl.truststore.location=E:\\temp\\client.truststore.jks
# ssl truststore.password is the password of the server certificate. This password is used for accessing the JKS file generated by Java.
ssl.truststore.password=dms@kafka
# ssl.endpoint.identification.algorithm indicates whether to verify the certificate domain name. This parameter must be left blank, which indicates disabling domain name verification.
ssl.endpoint.identification.algorithm=
```

- Consumer configuration file (the **dms.sdk.consumer.properties** file in the **message consumption code**)

  The information in bold is specific to different Kafka instances and must be modified. Other parameters can also be added.

```
#The topic name is in the specific production and consumption code.
########################
#Information about Kafka brokers. ip:port are the connection addresses and ports used by the instance. The values can be obtained by referring to the "Collecting Connection Information" section. Example: bootstrap.servers=100.xxx.xxx.87:909x,100.xxx.xxx.69:909x,100.xxx.xxx.155:909x
bootstrap.servers=ip1:port1,ip2:port2,ip3:port3
#Unique string to identify the group of consumer processes to which the consumer belongs. Configuring the same group.id for different processes indicates that the processes belong to the same consumer group.
group.id=1
#Method of turning the key into bytes
key.deserializer=org.apache.kafka.common.serialization.StringDeserializer
#Method of turning the value into bytes
value.deserializer=org.apache.kafka.common.serialization.StringDeserializer
#Offset reset policy
auto.offset.reset=earliest
########################
#Comment out the following parameters if SASL is not enabled.
########################
# Set the SASL authentication mechanism, username, and password.
#sasl.mechanism is the SASL mechanism. username and password are the SASL username and password. Obtain them by referring to section "Collecting Connection Information". For security purposes, you are advised to encrypt the username and password.
# If the SASL mechanism is PLAIN, the configuration is as follows:
sasl.mechanism=PLAIN
sasl.jaas.config=org.apache.kafka.common.security.plain.PlainLoginModule required \
    username="username" \
    password="password";
```



```
# If the SASL mechanism is SCRAM-SHA-512, the configuration is as follows:
sasl.mechanism=SCRAM-SHA-512
sasl.jaas.config=org.apache.kafka.common.security.scram.ScramLoginModule required \
    username="username" \
    password="password";

# Set security.protocol.
# If the security protocol is SASL_SSL, the configuration is as follows:
security.protocol=SASL_SSL
# ssl truststore.location is the path for storing the SSL certificate. The following code uses the path format in Windows as an example. Change the path format based on the actual running environment.
ssl.truststore.location=E:\\temp\\client.truststore.jks
# ssl truststore.password is the password of the server certificate. This password is used for accessing the JKS file generated by Java.
ssl.truststore.password=dms@kafka
# ssl.endpoint.identification.algorithm indicates whether to verify the certificate domain name. This parameter must be left blank, which indicates disabling domain name verification.
ssl.endpoint.identification.algorithm=
```

Producing Messages

- Test code

```
package com.dms.producer;

import org.apache.kafka.clients.producer.Callback;
import org.apache.kafka.clients.producer.RecordMetadata;
import org.junit.Test;

public class DmsProducerTest {
    @Test
    public void testProducer() throws Exception {
        DmsProducer<String, String> producer = new DmsProducer<String, String>();
        int partition = 0;
        try {
            for (int i = 0; i < 10; i++) {
                String key = null;
                String data = "The msg is " + i;
                //Enter the name of the topic you created. There are multiple APIs for producing messages. For details, see the Kafka official website or the following code.
                producer.produce("topic-0", partition, key, data, new Callback() {
                    public void onCompletion(RecordMetadata metadata,
                        Exception exception) {
                        if (exception != null) {
                            exception.printStackTrace();
                            return;
                        }
                        System.out.println("produce msg completed");
                    }
                });
                System.out.println("produce msg:" + data);
            }
        } catch (Exception e) {
            // TODO: Exception handling
            e.printStackTrace();
        } finally {
            producer.close();
        }
    }
}
```

- Message production code

```
package com.dms.producer;

import java.io.BufferedInputStream;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStream;
import java.net.URL;
```



```
import java.util.ArrayList;
import java.util.Enumeration;
import java.util.List;
import java.util.Properties;

import org.apache.kafka.clients.producer.Callback;
import org.apache.kafka.clients.producer.KafkaProducer;
import org.apache.kafka.clients.producer.Producer;
import org.apache.kafka.clients.producer.ProducerRecord;

public class DmsProducer<K, V> {
    //Add the producer configurations that have been specified earlier.
    public static final String CONFIG_PRODUCER_FILE_NAME = "dms.sdk.producer.properties";

    private Producer<K, V> producer;

    DmsProducer(String path)
    {
        Properties props = new Properties();
        try {
            InputStream in = new BufferedInputStream(new FileInputStream(path));
            props.load(in);
        }catch (IOException e)
        {
            e.printStackTrace();
            return;
        }
        producer = new KafkaProducer<K,V>(props);
    }
    DmsProducer()
    {
        Properties props = new Properties();
        try {
            props = loadFromClasspath(CONFIG_PRODUCER_FILE_NAME);
        }catch (IOException e)
        {
            e.printStackTrace();
            return;
        }
        producer = new KafkaProducer<K,V>(props);
    }

    /**
     * Producing messages
     *
     * @param topic        Topic
     * @param partition    partition
     * @param key          Message key
     * @param data         Message data
     */
    public void produce(String topic, Integer partition, K key, V data)
    {
        produce(topic, partition, key, data, null, (Callback)null);
    }

    /**
     * Producing messages
     *
     * @param topic        Topic
     * @param partition    partition
     * @param key          Message key
     * @param data         Message data
     * @param timestamp    timestamp
     */
    public void produce(String topic, Integer partition, K key, V data, Long timestamp)
    {
        produce(topic, partition, key, data, timestamp, (Callback)null);
    }
    /**
```



```
    * Producing messages
    *
    * @param topic        Topic
    * @param partition    partition
    * @param key          Message key
    * @param data         Message data
    * @param callback     callback
    */
   public void produce(String topic, Integer partition, K key, V data, Callback callback)
   {
       produce(topic, partition, key, data, null, callback);
   }

   public void produce(String topic, V data)
   {
       produce(topic, null, null, data, null, (Callback)null);
   }

   /**
    * Producing messages
    *
    * @param topic        Topic
    * @param partition    partition
    * @param key          Message key
    * @param data         Message data
    * @param timestamp    timestamp
    * @param callback     callback
    */
   public void produce(String topic, Integer partition, K key, V data, Long timestamp, Callback
callback)
   {
       ProducerRecord<K, V> kafkaRecord =
               timestamp == null ? new ProducerRecord<K, V>(topic, partition, key, data)
                       : new ProducerRecord<K, V>(topic, partition, timestamp, key, data);
       produce(kafkaRecord, callback);
   }

   public void produce(ProducerRecord<K, V> kafkaRecord)
   {
       produce(kafkaRecord, (Callback)null);
   }

   public void produce(ProducerRecord<K, V> kafkaRecord, Callback callback)
   {
       producer.send(kafkaRecord, callback);
   }

   public void close()
   {
       producer.close();
   }

   /**
    * get classloader from thread context if no classloader found in thread
    * context return the classloader which has loaded this class
    *
    * @return classloader
    */
   public static ClassLoader getCurrentClassLoader()
   {
       ClassLoader classLoader = Thread.currentThread()
               .getContextClassLoader();
       if (classLoader == null)
       {
           classLoader = DmsProducer.class.getClassLoader();
       }
       return classLoader;
   }
```



```
    /**
     * Load configuration information from classpath.
     *
     * @param configFileName Configuration file name
     * @return Configuration information
     * @throws IOException
     */
    public static Properties loadFromClasspath(String configFileName) throws IOException
    {
        ClassLoader classLoader = getCurrentClassLoader();
        Properties config = new Properties();

        List<URL> properties = new ArrayList<URL>();
        Enumeration<URL> propertyResources = classLoader
                .getResources(configFileName);
        while (propertyResources.hasMoreElements())
        {
            properties.add(propertyResources.nextElement());
        }

        for (URL url : properties)
        {
            InputStream is = null;
            try
            {
                is = url.openStream();
                config.load(is);
            }
            finally
            {
                if (is != null)
                {
                    is.close();
                    is = null;
                }
            }
        }

        return config;
    }
}
```

Consuming Messages

- Test code

```
package com.dms.consumer;

import org.apache.kafka.clients.consumer.ConsumerRecord;
import org.apache.kafka.clients.consumer.ConsumerRecords;
import org.junit.Test;
import java.util.Arrays;

public class DmsConsumerTest {
    @Test
    public void testConsumer() throws Exception {
        DmsConsumer consumer = new DmsConsumer();
        consumer.consume(Arrays.asList("topic-0"));
        try {
            for (int i = 0; i < 10; i++){
                ConsumerRecords<Object, Object> records = consumer.poll(1000);
                System.out.println("the numbers of topic:" + records.count());
                for (ConsumerRecord<Object, Object> record : records)
                {
                    System.out.println(record.toString());
                }
            }
        }catch (Exception e)
        {
            // TODO: Exception handling
```



```
            e.printStackTrace();
        }finally {
            consumer.close();
        }
    }
}
```

- Message consumption code

```
package com.dms.consumer;

import org.apache.kafka.clients.consumer.ConsumerRecords;
import org.apache.kafka.clients.consumer.KafkaConsumer;
import java.io.BufferedInputStream;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStream;
import java.net.URL;
import java.util.*;

public class DmsConsumer {

    public static final String CONFIG_CONSUMER_FILE_NAME = "dms.sdk.consumer.properties";

    private KafkaConsumer<Object, Object> consumer;

    DmsConsumer(String path)
    {
        Properties props = new Properties();
        try {
            InputStream in = new BufferedInputStream(new FileInputStream(path));
            props.load(in);
        }catch (IOException e)
        {
            e.printStackTrace();
            return;
        }
        consumer = new KafkaConsumer<Object, Object>(props);
    }

    DmsConsumer()
    {
        Properties props = new Properties();
        try {
            props = loadFromClasspath(CONFIG_CONSUMER_FILE_NAME);
        }catch (IOException e)
        {
            e.printStackTrace();
            return;
        }
        consumer = new KafkaConsumer<Object, Object>(props);
    }
    public void consume(List topics)
    {
        consumer.subscribe(topics);
    }

    public ConsumerRecords<Object, Object> poll(long timeout)
    {
        return consumer.poll(timeout);
    }

    public void close()
    {
        consumer.close();
    }

    /**
     * get classloader from thread context if no classloader found in thread
     * context return the classloader which has loaded this class
     *
```

```
     * @return classloader
     */
    public static ClassLoader getCurrentClassLoader()
    {
        ClassLoader classLoader = Thread.currentThread()
                .getContextClassLoader();
        if (classLoader == null)
        {
            classLoader = DmsConsumer.class.getClassLoader();
        }
        return classLoader;
    }

    /**
     * Load configuration information from classpath.
     *
     * @param configFileName Configuration file name
     * @return Configuration information
     * @throws IOException
     */
    public static Properties loadFromClasspath(String configFileName) throws IOException
    {
        ClassLoader classLoader = getCurrentClassLoader();
        Properties config = new Properties();

        List<URL> properties = new ArrayList<URL>();
        Enumeration<URL> propertyResources = classLoader
                .getResources(configFileName);
        while (propertyResources.hasMoreElements())
        {
            properties.add(propertyResources.nextElement());
        }

        for (URL url : properties)
        {
            InputStream is = null;
            try
            {
                is = url.openStream();
                config.load(is);
            }
            finally
            {
                if (is != null)
                {
                    is.close();
                    is = null;
                }
            }
        }

        return config;
    }
}
```

3.2 Setting Up the Java Development Environment

With the information collected in **Collecting Connection Information** and the network environment prepared for Kafka clients, you can proceed to configuring Kafka clients. This section describes how to configure Kafka clients to produce and consume messages.

Preparing Tools

- Maven

Apache Maven 3.0.3 or later can be downloaded from the **Maven official website**.

- JDK

  Java Development Kit 1.8.111 or later can be downloaded from the **Oracle official website**.

  After the installation, configure the Java environment variables.

- IntelliJ IDEA

  IntelliJ IDEA can be downloaded from the **IntelliJ IDEA official website** and be installed.

Procedure

Step 1 Download the **demo package**..

Decompress the package to obtain the following files.

Table 3-1 Files in the demo package

File	Directory	Description
DmsConsumer.java	.\src\main\java\com\dms\consumer	API for consuming messages
DmsProducer.java	.\src\main\java\com\dms\producer	API for producing messages
dms.sdk.consumer.properties	.\src\main\resources	Configuration information for consuming messages
dms.sdk.producer.properties	.\src\main\resources	Configuration information for producing messages
client.truststore.jks	.\src\main\resources	SSL certificate, used for SASL_SSL connection
DmsConsumerTest.java	.\src\test\java\com\dms\consumer	Test code of consuming messages
DmsProducerTest.java	.\src\test\java\com\dms\producer	Test code of producing messages
pom.xml	.\	Maven configuration file, containing the Kafka client dependencies

Step 2 In IntelliJ IDEA, import the demo project.

The demo project is a Java project built in Maven. Therefore, you need the JDK and the Maven plugin in IDEA.

Figure 3-1 Click **Import Project**.

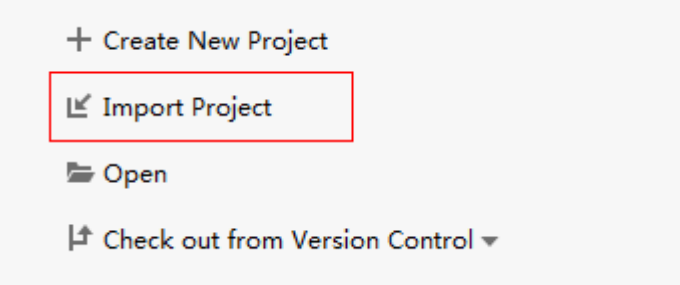


Figure 3-2 Choose **Maven**.

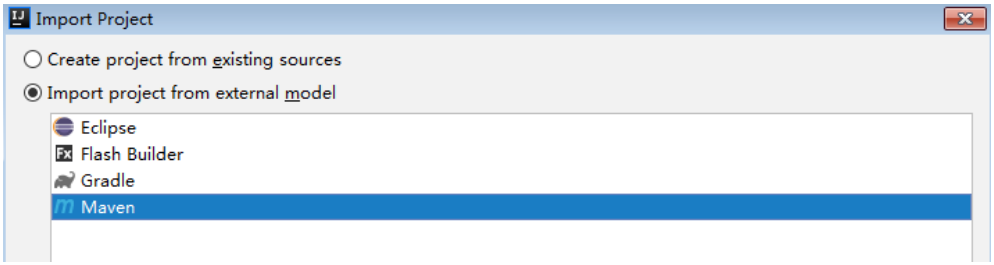
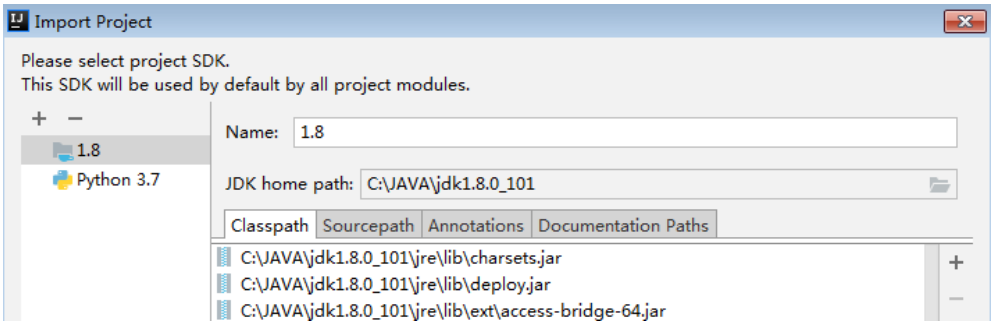
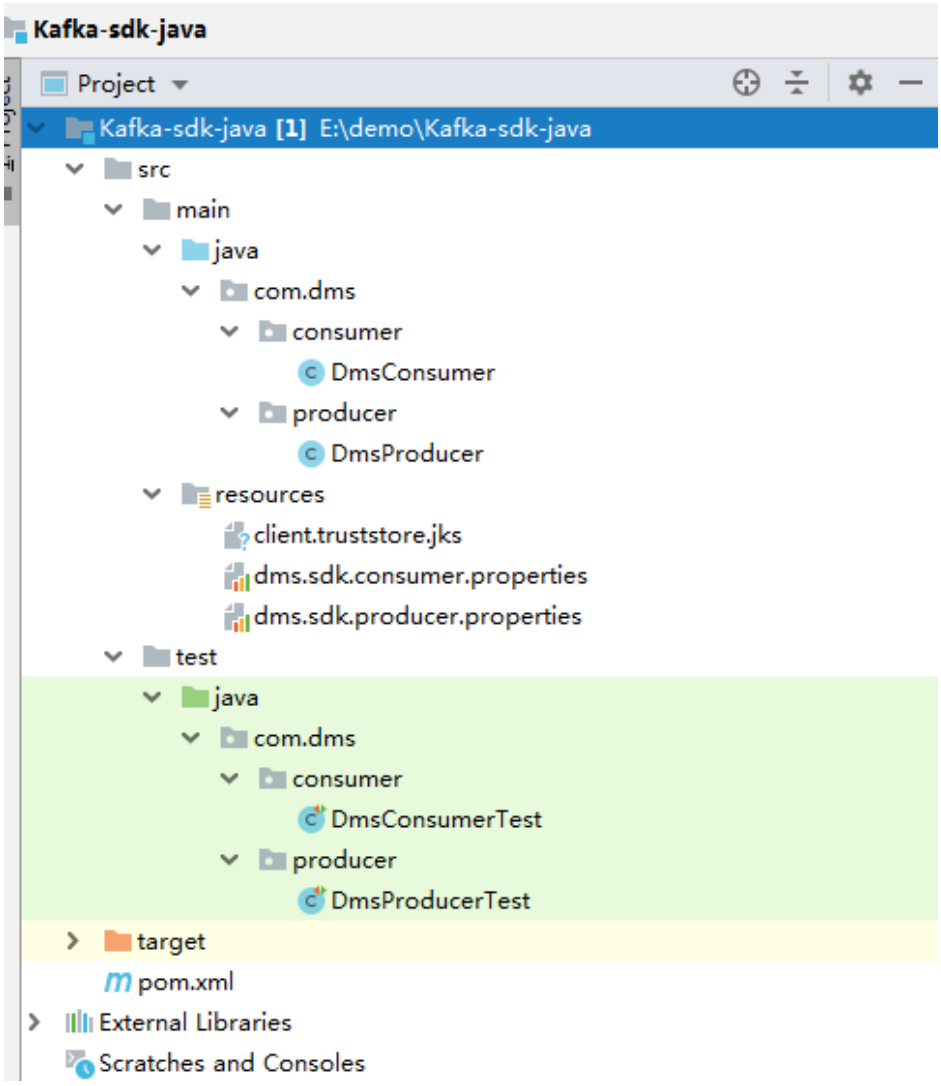


Figure 3-3 Select the JDK.



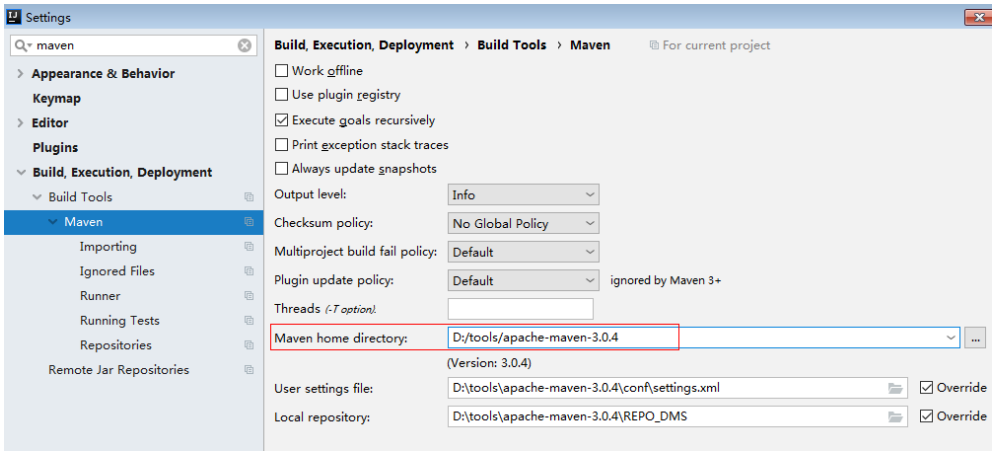
You can select other options or retain the default settings. Click **Finish**.

The demo project has been imported.



Step 3 Configure Maven.

Choose **Files** > **Settings**, set **Maven home directory** correctly, and select the required **settings.xml** file.

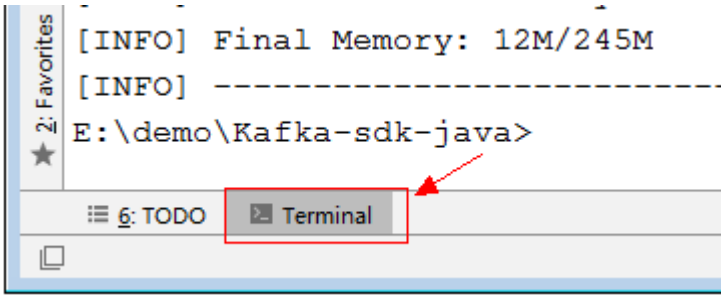


Step 4 Specify Kafka configurations.

This section uses the example of producing messages. For details, see **Producer configuration file**.

Step 5 In the down left corner of IDEA, click **Terminal**. In terminal, run the **mvn test** command to see how the demo project goes.

Figure 3-4 Opening terminal in IDEA



The following information is displayed for the producer:

```
-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running com.dms.producer.DmsProducerTest
produce msg:The msg is 0
produce msg:The msg is 1
produce msg:The msg is 2
produce msg:The msg is 3
produce msg:The msg is 4
produce msg:The msg is 5
produce msg:The msg is 6
produce msg:The msg is 7
produce msg:The msg is 8
produce msg:The msg is 9
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 138.877 sec
```

The following information is displayed for the consumer:

```
-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running com.dms.consumer.DmsConsumerTest
the numbers of topic:0
the numbers of topic:0
the numbers of topic:6
ConsumerRecord(topic = topic-0, partition = 2, offset = 0, CreateTime = 1557059377179, serialized key size = -1, serialized value size = 12, headers = RecordHeaders(headers = [], isReadOnly = false), key = null, value = The msg is 2)
ConsumerRecord(topic = topic-0, partition = 2, offset = 1, CreateTime = 1557059377195, serialized key size = -1, serialized value size = 12, headers = RecordHeaders(headers = [], isReadOnly = false), key = null, value = The msg is 5)
```

----End

4 Python

This section takes Linux CentOS as an example to describe how to access a Kafka instance using a Kafka client in Python, including how to install the client, and produce and consume messages.

Before getting started, ensure that you have collected the information listed in **Collecting Connection Information**.

Preparing the Environment

- Python

  Generally, Python is pre-installed in the system. You can enter **python** or **python3** in the command line to check whether Python has been installed. The **python** command checks whether python 2.x has been installed and the **python3** command checks whether python 3.x has been installed. Try both when you are not sure of the version.

  For example, if the following information is displayed, Python 3.x has been installed:

  ```
  [root@ecs-test python-kafka]# python3
  Python 3.7.1 (default, Jul  5 2020, 14:37:24)
  [GCC 4.8.5 20150623 (Red Hat 4.8.5-39)] on linux
  Type "help", "copyright", "credits" or "license" for more information.
  >>>
  ```

  If Python is not installed, run the following command:

  yum install python

- Kafka clients in Python

  Run the following command to install a Python client of the recommended version:

  - Python 2.x: **pip install kafka-python==2.0.1**
  - Python 3.x: **pip3 install kafka-python==2.0.1**

Producing Messages

Step 1 Create a file on the client to store the message production sample code.

```
touch producer.py
```

producer.py indicates a file name, which can be customized.

Step 2 Run the following command to edit the file:

```
vim producer.py
```

Step 3 Write the following sample code into the file and save.

- With SASL

```
from kafka import KafkaProducer
import ssl
##Connection information
conf = {
    'bootstrap_servers': ["ip1:port1","ip2:port2","ip3:port3"],
    'topic_name': 'topic_name',
    'sasl_username': 'username',
    'sasl_password': 'password'
}

context = ssl.create_default_context()
context = ssl.SSLContext(ssl.PROTOCOL_SSLv23)

context.verify_mode = ssl.CERT_REQUIRED
## The certificate file. Obtain the SSL certificate by referring to "Collecting Connection Information".
context.load_verify_locations("phy_ca.crt")

print('start producer')
producer = KafkaProducer(bootstrap_servers=conf['bootstrap_servers'],
                         sasl_mechanism="PLAIN",
                         ssl_context=context,
                         security_protocol='SASL_SSL',
                         sasl_plain_username=conf['sasl_username'],
                         sasl_plain_password=conf['sasl_password'])

data = bytes("hello kafka!", encoding="utf-8")
producer.send(conf['topic_name'], data)
producer.close()
print('end producer')
```

The parameters in the example code are described as follows. For details about how to obtain the parameter values, see **Collecting Connection Information**.

  - **bootstrap_servers**: instance connection address and port
  - **topic_name**: topic name
  - **sasl_plain_username/sasl_plain_password**: The username and password required to enable SASL_SSL, or the ones set in user creation. For security purposes, you are advised to encrypt the username and password.
  - **context.load_verify_locations**: certificate file. CRT certificates are used for accessing instances in Python.
  - **sasl_mechanism**: SASL authentication mechanism. View it on the **Basic Information** page of the Kafka instance console. If both SCRAM-SHA-512 and PLAIN are enabled, use either of them in connection configurations. For instances that were created much earlier, if **SASL Mechanism** is not displayed on the instance details page, PLAIN is used by default.

- Without SASL

```
from kafka import KafkaProducer

conf = {
    'bootstrap_servers': ["ip1:port1","ip2:port2","ip3:port3"],
    'topic_name': 'topic-name',
}

print('start producer')
producer = KafkaProducer(bootstrap_servers=conf['bootstrap_servers'])

data = bytes("hello kafka!", encoding="utf-8")
```



```
producer.send(conf['topic_name'], data)
producer.close()
print('end producer')
```

The parameters in the example code are described as follows. For details about how to obtain the parameter values, see **Collecting Connection Information**.

- **bootstrap_servers**: instance connection address and port
- **topic_name**: topic name

Step 4 Run the following command to run the sample code.

```
# Python 2.x
python producer.py

# Python 3.x
python3 producer.py
```

The following information will be displayed after the command is successfully executed.

```
[root@ecs-test ~]# python3 producer.py
start producer
end producer
[root@ecs-test ~]#
```

----End

Consuming Messages

Step 1 Create a file on the client to store the message consumption sample code.

```
touch consumer.py
```

consumer.py indicates a file name, which can be customized.

Step 2 Run the following command to edit the file:

```
vim consumer.py
```

Step 3 Write the following sample code into the file and save.

- With SASL

```
from kafka import KafkaConsumer
import ssl
##Connection information
conf = {
    'bootstrap_servers': ["ip1:port1","ip2:port2","ip3:port3"],
    'topic_name': 'topic_name',
    'sasl_username': 'username',
    'sasl_password': 'password',
    'consumer_id': 'consumer_id'
}

context = ssl.create_default_context()
context = ssl.SSLContext(ssl.PROTOCOL_SSLv23)

context.verify_mode = ssl.CERT_REQUIRED
## The certificate file. Obtain the SSL certificate by referring to "Collecting Connection Information".
context.load_verify_locations("phy_ca.crt")

print('start consumer')
consumer = KafkaConsumer(conf['topic_name'],
                         bootstrap_servers=conf['bootstrap_servers'],
                         group_id=conf['consumer_id'],
                         sasl_mechanism="PLAIN",
                         ssl_context=context,
                         security_protocol='SASL_SSL',
```



```
                    sasl_plain_username=conf['sasl_username'],
                    sasl_plain_password=conf['sasl_password'])

for message in consumer:
    print("%s:%d:%d: key=%s value=%s" % (message.topic, message.partition,message.offset,
message.key,message.value))

print('end consumer')
```

The parameters in the example code are described as follows. For details about how to obtain the parameter values, see **Collecting Connection Information**.

- **bootstrap_servers**: instance connection address and port
- **topic_name**: topic name
- **sasl_plain_username/sasl_plain_password**: The username and password required to enable SASL_SSL, or the ones set in user creation. For security purposes, you are advised to encrypt the username and password.
- **consumer_id**: custom consumer group name. If the specified consumer group does not exist, Kafka automatically creates one.
- **context.load_verify_locations**: certificate file. CRT certificates are used for accessing instances in Python.
- **sasl_mechanism**: SASL authentication mechanism. View it on the **Basic Information** page of the Kafka instance console. If both SCRAM-SHA-512 and PLAIN are enabled, use either of them in connection configurations. For instances that were created much earlier, if **SASL Mechanism** is not displayed on the instance details page, PLAIN is used by default.

- Without SASL

```
from kafka import KafkaConsumer

conf = {
    'bootstrap_servers': ["ip1:port1","ip2:port2","ip3:port3"],
    'topic_name': 'topic-name',
    'consumer_id': 'consumer-id'
}

print('start consumer')
consumer = KafkaConsumer(conf['topic_name'],
                         bootstrap_servers=conf['bootstrap_servers'],
                         group_id=conf['consumer_id'])

for message in consumer:
    print("%s:%d:%d: key=%s value=%s" % (message.topic, message.partition,message.offset,
message.key,message.value))

print('end consumer')
```

The parameters in the example code are described as follows. For details about how to obtain the parameter values, see **Collecting Connection Information**.

- **bootstrap_servers**: instance connection address and port
- **topic_name**: topic name
- **consumer_id**: custom consumer group name. If the specified consumer group does not exist, Kafka automatically creates one.

Step 4 Run the following command to run the sample code.

```
# Python 2.x
python consumer.py
```



```
# Python 3.x
python3 consumer.py
```

The following information will be displayed after the command is successfully executed.

```
[root@ecs-test ~]# python3 consumer.py
start consumer
```

Press **Ctrl+C** to cancel.

----End

5 Go

This section takes Linux CentOS as an example to describe how to access a Kafka instance using a Kafka client in Go 1.16.5, including how to obtain the demo code, and produce and consume messages.

Before getting started, ensure that you have collected the information listed in **Collecting Connection Information**.

Preparing the Environment

- Run the following command to check whether Go has been installed:
  ```
  go version
  ```
  If the following information is displayed, Go has been installed.
  ```
  [root@ecs-test confluent-kafka-go]# go version
  go version go1.16.5 linux/amd64
  ```
  If Go is not installed, do as follows to install it:
  ```
  # Download the Go installation package.
  wget https://go.dev/dl/go1.16.5.linux-amd64.tar.gz

  # Decompress it to the /usr/local directory. The /usr/local directory can be changed as required.
  sudo tar -C /usr/local -xzf go1.16.5.linux-amd64.tar.gz

  # Set the environment variable.
  echo 'export PATH=$PATH:/usr/local/go/bin' >> ~/.profile
  source ~/.profile
  ```
- Run the following command to obtain the code used in the demo:
  ```
  go get github.com/confluentinc/confluent-kafka-go/kafka
  ```

Producing Messages

- With SASL
  ```
  package main

  import (
      "bufio"
      "fmt"
      "github.com/confluentinc/confluent-kafka-go/kafka"
      "log"
      "os"
      "os/signal"
      "syscall"
  )

  var (
  ```

```
    brokers  = "ip1:port1,ip2:port2,ip3:port3"
    topics   = "topic_name"
    user     = "username"
    password = "password"
    caFile   = "phy_ca.crt"     // Obtain the SSL certificate by referring to section "Collecting
Connection Information".
)

func main() {
    log.Println("Starting a new kafka producer")

    config := &kafka.ConfigMap{
        "bootstrap.servers": brokers,
        "security.protocol": "SASL_SSL",
        "sasl.mechanism":    "PLAIN",
        "sasl.username":     user,
        "sasl.password":     password,
        "ssl.ca.location":   caFile,
        "ssl.endpoint.identification.algorithm": "none",
    }
    producer, err := kafka.NewProducer(config)
    if err != nil {
        log.Panicf("producer error, err: %v", err)
        return
    }

    go func() {
        for e := range producer.Events() {
            switch ev := e.(type) {
            case *kafka.Message:
                if ev.TopicPartition.Error != nil {
                    log.Printf("Delivery failed: %v\n", ev.TopicPartition)
                } else {
                    log.Printf("Delivered message to %v\n", ev.TopicPartition)
                }
            }
        }
    }()

    // Produce messages to topic (asynchronously)
    fmt.Println("please enter message:")
    go func() {
        for {
            err := producer.Produce(&kafka.Message{
                TopicPartition: kafka.TopicPartition{Topic: &topics, Partition: kafka.PartitionAny},
                Value:          GetInput(),
            }, nil)
            if err != nil {
                log.Panicf("send message fail, err: %v", err)
                return
            }
        }
    }()

    sigterm := make(chan os.Signal, 1)
    signal.Notify(sigterm, syscall.SIGINT, syscall.SIGTERM)
    select {
    case <-sigterm:
        log.Println("terminating: via signal")
    }
    // Wait for message deliveries before shutting down
    producer.Flush(15 * 1000)
    producer.Close()
}

func GetInput() []byte {
    reader := bufio.NewReader(os.Stdin)
    data, _, _ := reader.ReadLine()
```



```
    return data
}
```

The parameters in the example code are described as follows. For details about how to obtain the parameter values, see **Collecting Connection Information**.

- **brokers**: instance connection address and port
- **topics**: topic name
- **user/password**: The username and password required to enable SASL_SSL, or the ones set in user creation. For security purposes, you are advised to encrypt the username and password.
- **caFile**: certificate file
- **sasl.mechanism**: SASL authentication mechanism. View it on the **Basic Information** page of the Kafka instance console. If both SCRAM-SHA-512 and PLAIN are enabled, use either of them in connection configurations. For instances that were created much earlier, if **SASL Mechanism** is not displayed on the instance details page, PLAIN is used by default.

- Without SASL

```
package main

import (
    "bufio"
    "fmt"
    "github.com/confluentinc/confluent-kafka-go/kafka"
    "log"
    "os"
    "os/signal"
    "syscall"
)

var (
    brokers  = "ip1:port1,ip2:port2,ip3:port3"
    topics   = "topic_name"
)

func main() {
    log.Println("Starting a new kafka producer")

    config := &kafka.ConfigMap{
        "bootstrap.servers": brokers,
    }
    producer, err := kafka.NewProducer(config)
    if err != nil {
        log.Panicf("producer error, err: %v", err)
        return
    }

    go func() {
        for e := range producer.Events() {
            switch ev := e.(type) {
            case *kafka.Message:
                if ev.TopicPartition.Error != nil {
                    log.Printf("Delivery failed: %v\n", ev.TopicPartition)
                } else {
                    log.Printf("Delivered message to %v\n", ev.TopicPartition)
                }
            }
        }
    }()

    // Produce messages to topic (asynchronously)
    fmt.Println("please enter message:")
    go func() {
```



```
        for {
            err := producer.Produce(&kafka.Message{
                TopicPartition: kafka.TopicPartition{Topic: &topics, Partition: kafka.PartitionAny},
                Value:          GetInput(),
            }, nil)
            if err != nil {
                log.Panicf("send message fail, err: %v", err)
                return
            }
        }
    }()

    sigterm := make(chan os.Signal, 1)
    signal.Notify(sigterm, syscall.SIGINT, syscall.SIGTERM)
    select {
    case <-sigterm:
        log.Println("terminating: via signal")
    }
    // Wait for message deliveries before shutting down
    producer.Flush(15 * 1000)
    producer.Close()
}

func GetInput() []byte {
    reader := bufio.NewReader(os.Stdin)
    data, _, _ := reader.ReadLine()
    return data
}
```

The parameters in the example code are described as follows. For details about how to obtain the parameter values, see **Collecting Connection Information**.

- **brokers**: instance connection address and port
- **topics**: topic name

Consuming Messages

- With SASL

```
package main

import (
    "fmt"
    "github.com/confluentinc/confluent-kafka-go/kafka"
    "log"
    "os"
    "os/signal"
    "syscall"
)

var (
    brokers  = "ip1:port1,ip2:port2,ip3:port3"
    group    = "group-id"
    topics   = "topic_name"
    user     = "username"
    password = "password"
    caFile   = "phy_ca.crt"     // Obtain the SSL certificate by referring to section "Collecting Connection Information".
)

func main() {
    log.Println("Starting a new kafka consumer")

    config := &kafka.ConfigMap{
        "bootstrap.servers": brokers,
        "group.id":          group,
        "auto.offset.reset": "earliest",
        "security.protocol": "SASL_SSL",
```



```
        "sasl.mechanism":    "PLAIN",
        "sasl.username":     user,
        "sasl.password":     password,
        "ssl.ca.location":   caFile,
        "ssl.endpoint.identification.algorithm": "none",
    }

    consumer, err := kafka.NewConsumer(config)
    if err != nil {
        log.Panicf("Error creating consumer: %v", err)
        return
    }

    err = consumer.SubscribeTopics([]string{topics}, nil)
    if err != nil {
        log.Panicf("Error subscribe consumer: %v", err)
        return
    }

    go func() {
        for {
            msg, err := consumer.ReadMessage(-1)
            if err != nil {
                log.Printf("Consumer error: %v (%v)", err, msg)
            } else {
                fmt.Printf("Message on %s: %s\n", msg.TopicPartition, string(msg.Value))
            }
        }
    }()

    sigterm := make(chan os.Signal, 1)
    signal.Notify(sigterm, syscall.SIGINT, syscall.SIGTERM)
    select {
    case <-sigterm:
        log.Println("terminating: via signal")
    }
    if err = consumer.Close(); err != nil {
        log.Panicf("Error closing consumer: %v", err)
    }
}
```

The parameters in the example code are described as follows. For details about how to obtain the parameter values, see **Collecting Connection Information**.

- **brokers**: instance connection address and port
- **group**: custom consumer group name. If the specified consumer group does not exist, Kafka automatically creates one.
- **topics**: topic name
- **user/password**: The username and password required to enable SASL_SSL, or the ones set in user creation. For security purposes, you are advised to encrypt the username and password.
- **caFile**: certificate file
- **sasl.mechanism**: SASL authentication mechanism. View it on the **Basic Information** page of the Kafka instance console. If both SCRAM-SHA-512 and PLAIN are enabled, use either of them in connection configurations. For instances that were created much earlier, if **SASL Mechanism** is not displayed on the instance details page, PLAIN is used by default.

- Without SASL

```
package main

import (
    "fmt"
```



```
    "github.com/confluentinc/confluent-kafka-go/kafka"
    "log"
    "os"
    "os/signal"
    "syscall"
)

var (
    brokers  = "ip1:port1,ip2:port2,ip3:port3"
    group    = "group-id"
    topics   = "topic_name"
)

func main() {
    log.Println("Starting a new kafka consumer")

    config := &kafka.ConfigMap{
        "bootstrap.servers": brokers,
        "group.id":          group,
        "auto.offset.reset": "earliest",
    }

    consumer, err := kafka.NewConsumer(config)
    if err != nil {
        log.Panicf("Error creating consumer: %v", err)
        return
    }

    err = consumer.SubscribeTopics([]string{topics}, nil)
    if err != nil {
        log.Panicf("Error subscribe consumer: %v", err)
        return
    }

    go func() {
        for {
            msg, err := consumer.ReadMessage(-1)
            if err != nil {
                log.Printf("Consumer error: %v (%v)", err, msg)
            } else {
                fmt.Printf("Message on %s: %s\n", msg.TopicPartition, string(msg.Value))
            }
        }
    }()

    sigterm := make(chan os.Signal, 1)
    signal.Notify(sigterm, syscall.SIGINT, syscall.SIGTERM)
    select {
    case <-sigterm:
        log.Println("terminating: via signal")
    }
    if err = consumer.Close(); err != nil {
        log.Panicf("Error closing consumer: %v", err)
    }
}
```

The parameters in the example code are described as follows. For details about how to obtain the parameter values, see **Collecting Connection Information**.

- **brokers**: instance connection address and port
- **group**: custom consumer group name. If the specified consumer group does not exist, Kafka automatically creates one.
- **topics**: topic name

6 Obtaining an Open-Source Kafka Client

Kafka instances can be accessed on clients using multiple languages. For more information, see **Clients**.

The Kafka official website supports C/C++, Python, Go, PHP, and Node.js clients.

Kafka instances are fully compatible with open-source clients. You can access your instance as instructed by the official Kafka website.

A Change History

Date	Description
2023-05-26	This issue incorporates the following changes: • Added support for the SASL mechanism in sections **Collecting Connection Information**, **Configuring Kafka Clients in Java**, **Python**, and **Go**.
2022-12-30	This issue incorporates the following changes: • Supported Kafka 2.7. • Added **Go**.
2020-12-02	This issue is the first official release.

