**Specifications:**

**• 3 Servers**

**• Distribution:**

**Ubuntu 18.04 Bionic Beaver LTS**

**• Size: 1 Large, 2 Small**

1. On all three nodes, add the GPG key and package repository, then install Confluent and Java.

wget -qO - https://packages.confluent.io/deb/5.2/archive.key | sudo apt-key add -

sudo add-apt-repository "deb [arch=amd64] https://packages.confluent.io/deb/5.2 stable main"

sudo apt-get update && sudo apt-get install -y openjdk-8-jdk confluent-community-2.12

1. On all three nodes, edit the hosts file.

sudo nano /etc/hosts

1. Add the following entries to the hosts file on all three servers. Use the private IP addresses of your three servers (you can find them in Cloud Playground).

<server 1 private IP> zoo1

<server 1 private IP> zoo2

<server 1 private IP> zoo3

1. Edit the ZooKeeper config file.

sudo nano /etc/kafka/zookeeper.properties

1. Delete the contents of the config file, and replace them with the following:

tickTime=2000

dataDir=/var/lib/zookeeper/

clientPort=2181

initLimit=5

syncLimit=2

server.1=zoo1:2888:3888

server.2=zoo2:2888:3888

server.3=zoo3:2888:3888

autopurge.snapRetainCount=3

autopurge.purgeInterval=24

1. Set the Zookeeper ID for each server.

sudo nano /var/lib/zookeeper/myid

1. On each server, set the contents of /var/lib/zookeeper/myid to the server's ID. On Server 1, enter 1. On Server 2, enter 2, and on Server 3, enter 3.

<server id 1, 2, or 3>

1. Edit the Kafka config file.

sudo nano /etc/kafka/server.properties

1. Edit the broker.id, advertised.listeners, and zookeeper.connect in the config file. Set the broker ID to the appropriate ID for each server (1 on Server 1, 2 on Server 2, and so on).
2. For advertised.listeners, provide the hostname for each server: zoo1, zoo2, or zoo3 as appropriate.
3. Set zookeeper.connect to zoo1:2181.

broker.id=<server id 1, 2, or 3>

...

advertised.listeners=PLAINTEXT://<hostname zoo1, zoo2, or zoo3>:9092

...

zookeeper.connect=zoo1:2181

1. Start and enable the Zookeeper service.

sudo systemctl start confluent-zookeeper

sudo systemctl enable confluent-zookeeper

1. Wait a few seconds, then do the same for the Kafka service.

sudo systemctl start confluent-kafka

sudo systemctl enable confluent-kafka

1. Check the services to make sure they are running. Both services should be active (running) on all three servers.

sudo systemctl status confluent\*

1. Test your cluster by listing the current topics.

kafka-topics --list --bootstrap-server localhost:9092

1. Since you have not created any topics yet, you will only see a default topic. The output should look like this:

\_confluent.support.metrics

## **Kafka from the Command Line**

1. Download the Kafka binaries:
2. wget http://mirror.cogentco.com/pub/apache/kafka/2.2.0/kafka\_2.12-2.2.0.tgz
3. tar -xvf kafka\_2.12-2.2.0.tgz
4. mv kafka\_2.12-2.2.0 kafka

cd kafka/

1. Use the shell scripts that come with the Kafka binaries to list the topics in the cluster:

./bin/kafka-topics.sh --bootstrap-server localhost:9092 --list

1. List the shell scripts:

ls -l ./bin/

1. Use the kafka-topics command that comes with the Confluent package installation to list the topics in the cluster:

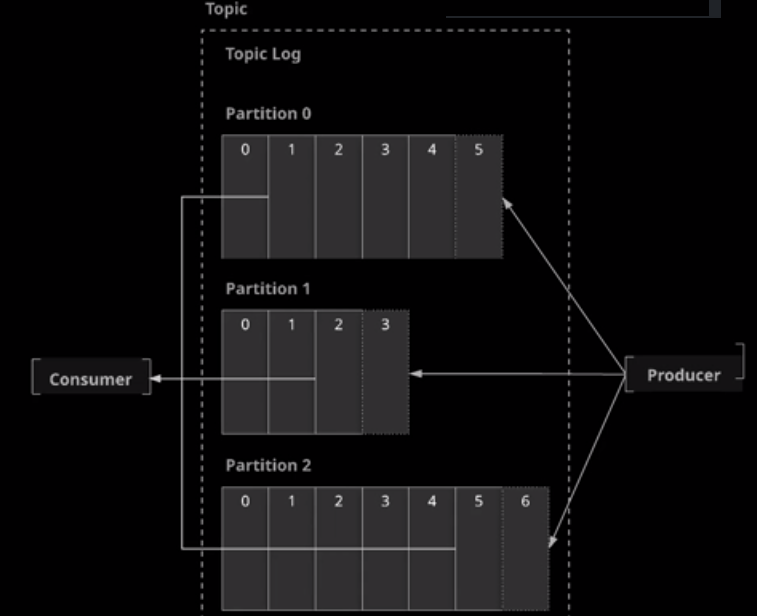
kafka-topics --bootstrap-server localhost:9092 --list

1. List the Kafka shell scripts that were installed with the Confluent packages:

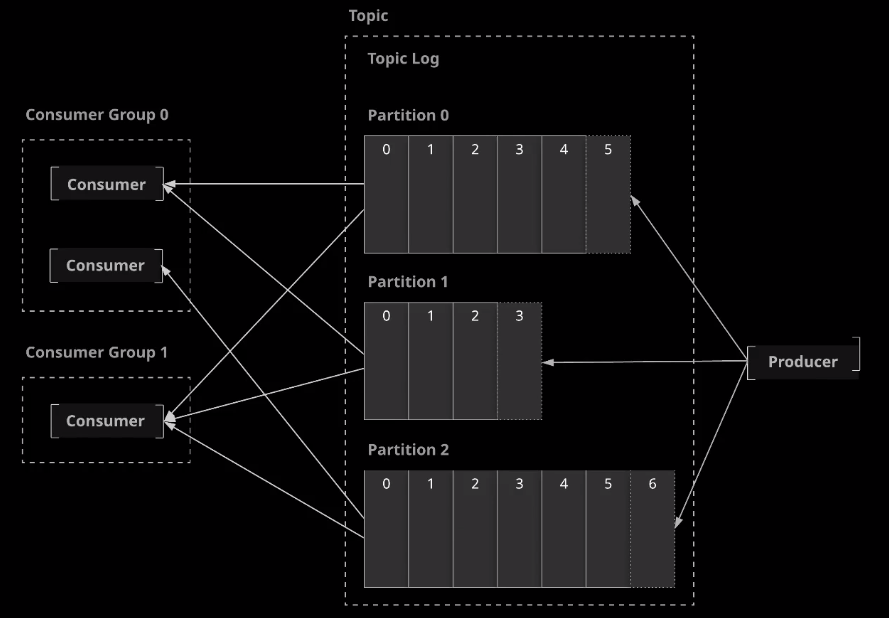
ls -l /usr/bin | grep kafka

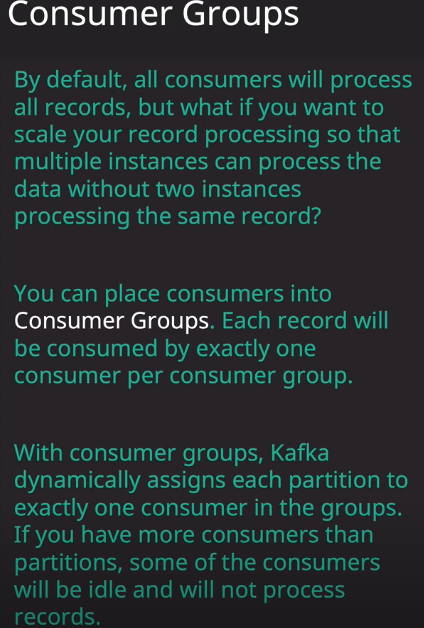
1. Run the confluent command:

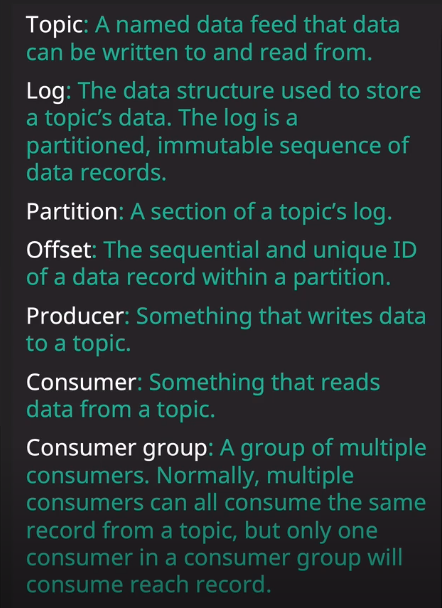
confluent

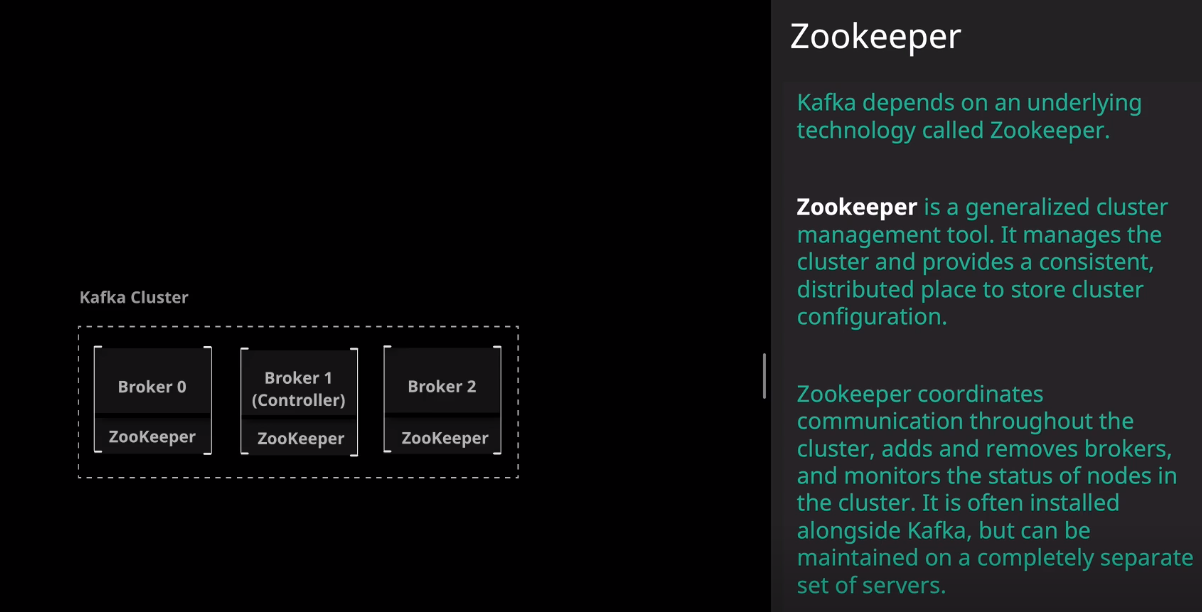


Any number of Consumer for a Topic is allowed



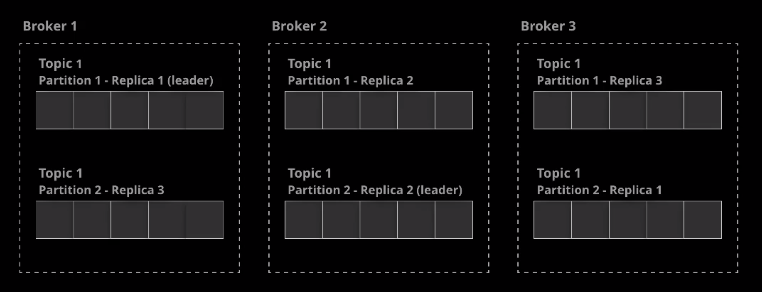






## **Partitions and Replication**

Part of Kafka's approach to fault tolerance involves maintaining replicas of topic partition data.



Replication Factor can’t be higher than Number of Brokers

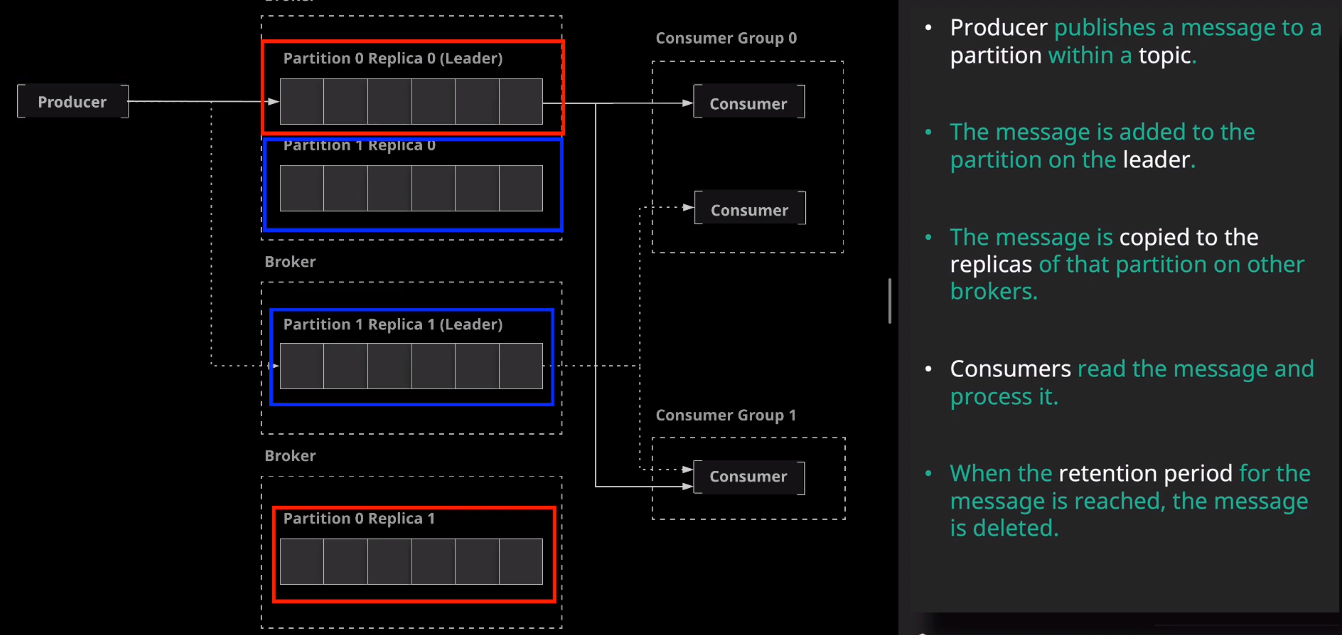
Create a topic with multiple partitions and a replication factor greater than 1:

kafka-topics --bootstrap-server localhost:9092 --create --topic my-topic --partitions 3 --replication-factor 2

Describe the topic to view information about its partitions and replicas:

kafka-topics --bootstrap-server localhost:9092 --describe --topic my-topic

**Bird View :**



# **Working with Kafka from the Command Line**

**Create a Kafka Topic for the Inventory Purchase Data**

1. Create the topic using the kafka-topics command:

kafka-topics --create --bootstrap-server localhost:9092 --replication-factor 3 --partitions 6 --topic inventory\_purchases

**Test the Setup by Publishing and Consuming Some Data**

1. Start a command line producer:

kafka-console-producer --broker-list localhost:9092 --topic inventory\_purchases

1. Type in a few lines of test data. Since we are working with merely test data, a specific format is not required. It could look like this:
2. product: apples, quantity: 5

product: lemons, quantity: 7

1. Once the test messages are published, we can exit the producer.
2. Start up a command line consumer:

kafka-console-consumer --bootstrap-server localhost:9092 --topic inventory\_purchases --from-beginning

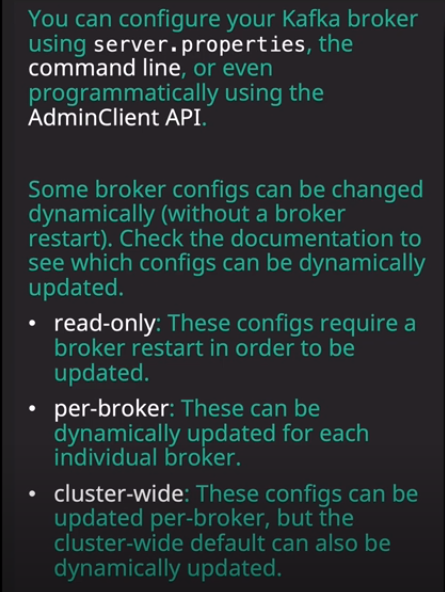
1. We should see the test messages that were published earlier:
2. product: apples, quantity: 5

product: lemons, quantity: 7

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_Advanced Configuration……………………………

**Broker Configuration**

#### **Broker Configs**



1. List the current configurations for broker 1.

kafka-configs --bootstrap-server localhost:9092 --entity-type brokers --entity-name 1 --describe

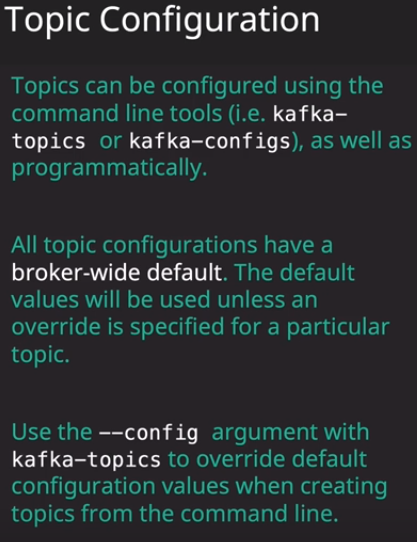
1. Modify the configuration for broker 1 by setting log.cleaner.threads to 2.

kafka-configs --bootstrap-server localhost:9092 --entity-type brokers --entity-name 1 --alter --add-config log.cleaner.threads=2

1. List the configurations for broker 1 to see the newly added configuration.

kafka-configs --bootstrap-server localhost:9092 --entity-type brokers --entity-name 1 --describe

**Topic Configurations:**



1. Create a new topic with a configuration override.

kafka-topics --bootstrap-server localhost:9092 --create --topic configured-topic --partitions 1 --replication-factor 1 --config max.message.bytes=64000

1. List the configurations for the topic to see the configuration override.

kafka-configs --zookeeper localhost:2181 --entity-type topics --entity-name configured-topic --describe

1. Modify the configuration override for the existing topic.

kafka-configs --zookeeper localhost:2181 --entity-type topics --entity-name configured-topic --alter --add-config max.message.bytes=65000

1. List the topic configurations again to see the changes.

kafka-configs --zookeeper localhost:2181 --entity-type topics --entity-name configured-topic --describe

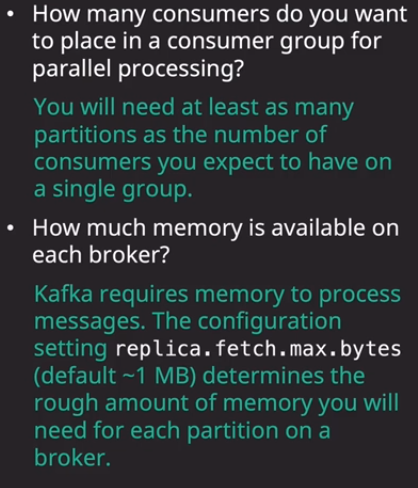
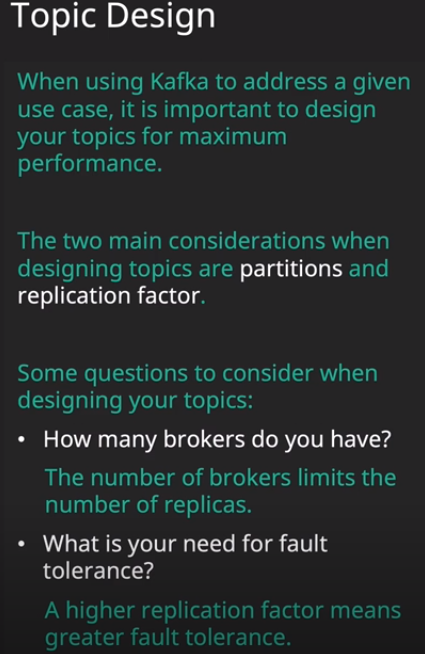
1. Modify a broker-wide default topic configuration.

kafka-configs --bootstrap-server localhost:9092 --entity-type brokers --entity-name 1 --alter --add-config message.max.bytes=66000

1. View the broker configuration to see the changes to the default.

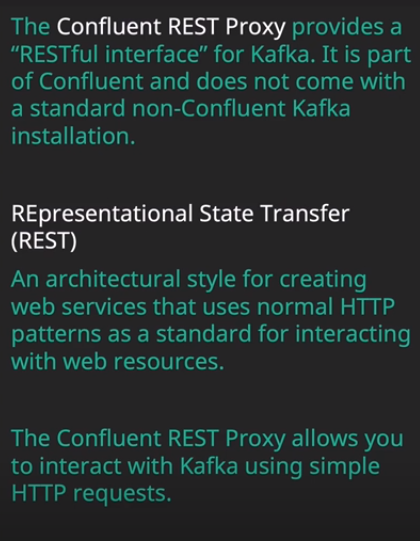
kafka-configs --bootstrap-server localhost:9092 --entity-type brokers --entity-name 1 --describe

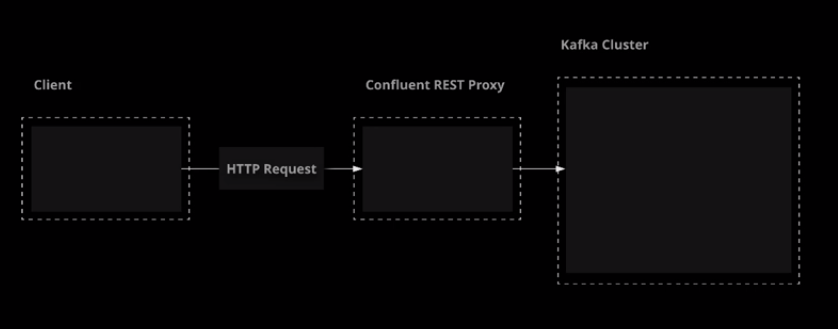
==========================================================================Topic Design =====================================



5 Consumers –2 partitions ….only two consumers will be active and you are not taking full advantage

======================Confluent-----





Start the confluent-schema-registry and confluent-kafka-rest services on your large broker.

sudo systemctl start confluent-schema-registry confluent-kafka-rest

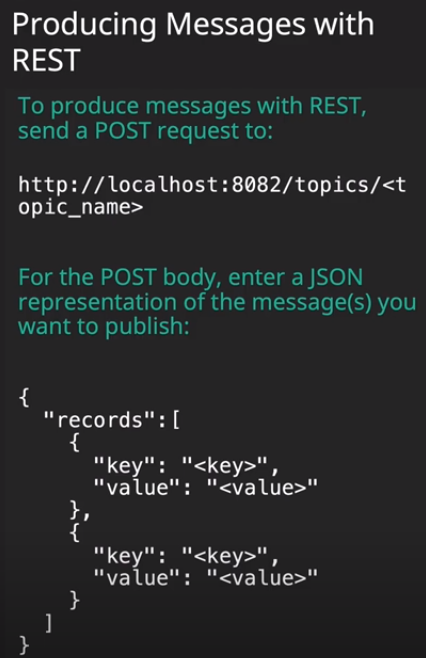
sudo systemctl enable confluent-schema-registry confluent-kafka-rest

Verify that the services are running.

sudo systemctl status confluent-schema-registry confluent-kafka-rest

<https://docs.confluent.io/current/kafka-rest/api.html>

## **Producing Messages with REST Proxy**



Create a topic to use for testing.

kafka-topics --bootstrap-server localhost:9092 --create --topic rest-test-topic --partitions 1 --replication-factor 1

Publish some messages to the topic using the Confluent REST proxy.

curl -X POST -H "Content-Type: application/vnd.kafka.json.v2+json" \

-H "Accept: application/vnd.kafka.v2+json" \

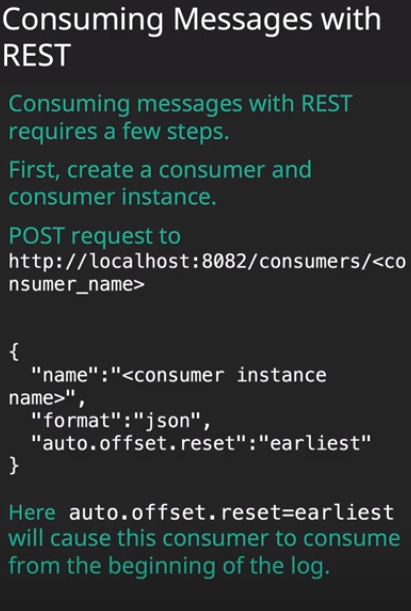
--data '{"records":[{"key":"message","value":"Hello"},{"key":"message","value":"World"}]}' "http://localhost:8082/topics/rest-test-topic"

You should get a response containing metadata about the two new messages.

Use a console consumer to verify that your messages are present in the topic.

kafka-console-consumer --bootstrap-server localhost:9092 --topic rest-test-topic --from-beginning --property print.key=true

## **Consuming Messages with REST Proxy**



1. Create a consumer and a consumer instance that will start from the beginning of the topic log.

curl -X POST -H "Content-Type: application/vnd.kafka.v2+json" \

--data '{"name": "my\_consumer\_instance", "format": "json", "auto.offset.reset": "earliest"}' \

http://localhost:8082/consumers/my\_json\_consumer

2 .Subscribe the consumer to the topic.

curl -X POST -H "Content-Type: application/vnd.kafka.v2+json" \

--data '{"topics":["rest-test-topic"]}' \

http://localhost:8082/consumers/my\_json\_consumer/instances/my\_consumer\_instance/subscription

3 .Consume the messages.

curl -X GET -H "Accept: application/vnd.kafka.json.v2+json" \

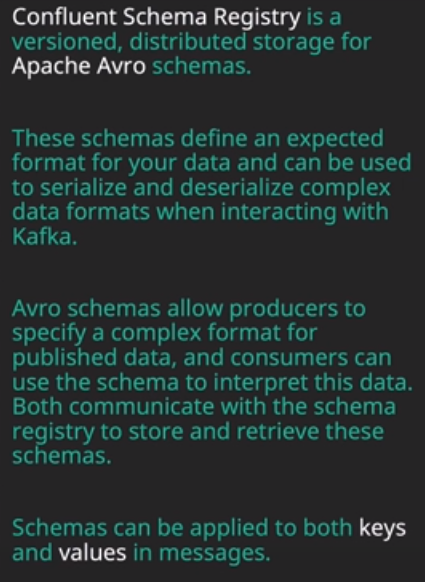
http://localhost:8082/consumers/my\_json\_consumer/instances/my\_consumer\_instance/records

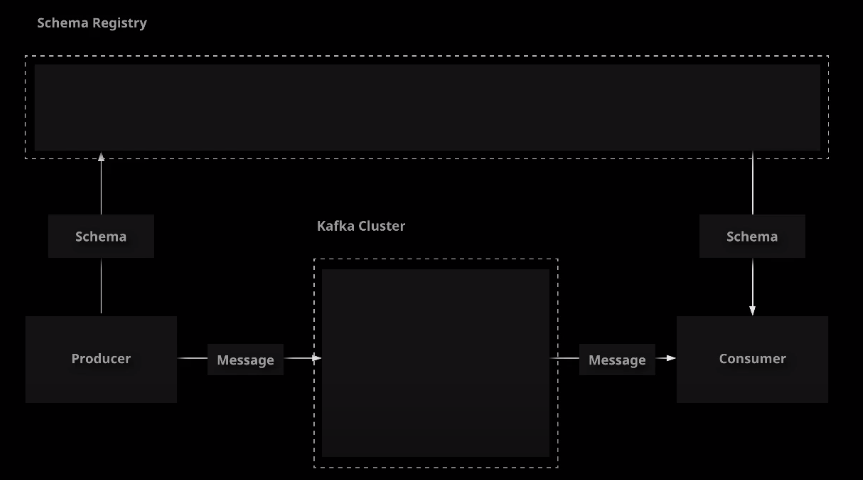
1. When you are finished using the consumer, close it to clean up.

curl -X DELETE -H "Content-Type: application/vnd.kafka.v2+json" \

http://localhost:8082/consumers/my\_json\_consumer/instances/my\_consumer\_instance

#### =============== **Confluent Schema Registry==========**

Confluent Schema Registry provides powerful functionality for maintaining and evolving contracts regarding data format between your producers and consumers



Registry acts as central place to store all of these Schemas with version at central location:

Broker 3 being Large Ubuntu

1. Make sure Confluent Schema Registry is running.

sudo systemctl status confluent-schema-registry

1. If you haven't started Schema Registry yet, you can do so like this:
2. sudo systemctl start confluent-schema-registry

sudo systemctl enable confluent-schema-registry

## **Creating an Avro Schema**

Avro schemas allow you to define your own custom data formats with multiple fields. You can then use these formats to serialize and deserialize Kafka records.

Clone the starter project.

cd ~/

git clone https://github.com/linuxacademy/content-ccdak-schema-registry.git

Create a schema definition file.

cd content-ccdak-schema-registry

mkdir -p src/main/avro/com/linuxacademy/ccdak/schemaregistry

nano src/main/avro/com/linuxacademy/ccdak/schemaregistry/Person.avsc

Implement a schema definition for data representing a person.

{

"namespace": "com.linuxacademy.ccdak.schemaregistry",

"type": "record",

"name": "Person",

"fields": [

{"name": "id", "type": "int"},

{"name": "first\_name", "type": "string"},

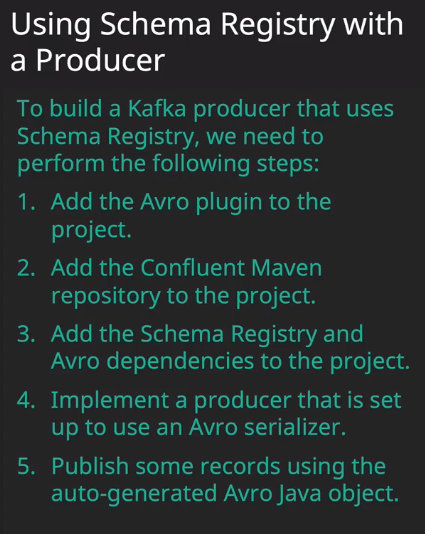
{"name": "last\_name", "type": "string"},

{"name": "email", "type": "string"}

]

}

## **Using Schema Registry with a Kafka Producer**



With Confluent Schema Registry, Kafka producers can register a schema with the registry and then use that schema to convert a Java object into data that can be published to a topic.

Edit build.gradle. to add pluggin

cd content-ccdak-schema-registry

nano build.gradle

Add the Confluent repository, Avro plugin, and Avro dependencies.

plugins {

id 'application'

id 'com.commercehub.gradle.plugin.avro' version '0.9.1'

}

repositories {

mavenCentral()

maven { url 'https://packages.confluent.io/maven' }

}

dependencies {

implementation 'org.apache.kafka:kafka-clients:2.2.1'

implementation 'io.confluent:kafka-avro-serializer:5.3.0'

implementation 'org.apache.avro:avro:1.9.0'

testImplementation 'junit:junit:4.12'

}

...

Edit the ProducerMain class.

nano src/main/java/com/linuxacademy/ccdak/schemaregistry/SchemaRegistryProducerMain.java

Implement a producer that serializes data using an Avro schema. This class uses the Person schema created in an earlier lesson at src/main/avro/com/linuxacademy/ccdak/schemaregistry/Person.avsc.

package com.linuxacademy.ccdak.schemaregistry;

import io.confluent.kafka.serializers.AbstractKafkaAvroSerDeConfig;

import io.confluent.kafka.serializers.KafkaAvroSerializer;

import java.util.Properties;

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.common.serialization.StringSerializer;

public class SchemaRegistryProducerMain {

public static void main(String[] args) {

final Properties props = new Properties();

props.put(ProducerConfig.BOOTSTRAP\_SERVERS\_CONFIG, "localhost:9092");

props.put(ProducerConfig.ACKS\_CONFIG, "all");

props.put(ProducerConfig.RETRIES\_CONFIG, 0);

props.put(ProducerConfig.KEY\_SERIALIZER\_CLASS\_CONFIG, StringSerializer.class);

props.put(ProducerConfig.VALUE\_SERIALIZER\_CLASS\_CONFIG, KafkaAvroSerializer.class);

props.put(AbstractKafkaAvroSerDeConfig.SCHEMA\_REGISTRY\_URL\_CONFIG, "http://localhost:8081");

KafkaProducer<String, Person> producer = new KafkaProducer<String, Person>(props);

Person kenny = new Person(125745, "Kenny", "Armstrong", "kenny@linuxacademy.com");

producer.send(new ProducerRecord<String, Person>("employees", kenny.getId().toString(), kenny));

Person terry = new Person(943256, "Terry", "Cox", "terry@linuxacademy.com");

producer.send(new ProducerRecord<String, Person>("employees", terry.getId().toString(), terry));

producer.close();

}

}

Create the employees topic to use for testing.

kafka-topics --bootstrap-server localhost:9092 --create --topic employees --partitions 1 --replication-factor 1

Run your code.

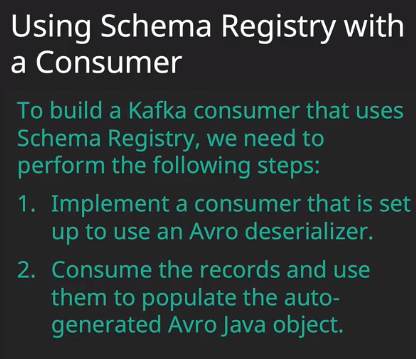
./gradlew runProducer

Verify that the messages published by the producer are present in the topic.

kafka-console-consumer --bootstrap-server localhost:9092 --topic employees --from-beginning

Note that the data will not display correctly since the console consumer's default string deserializer is not set up to correctly interpret the serialized data. However, you can still use the console consumer to verify that the data is present.

## **Using Schema Registry With a Kafka Consume**



When a producer uses Confluent Schema Registry and publishes data to a topic, consumers can use Schema Registry to download the schema and properly deserialize the data.

Edit the Consumer Main class.

cd content-ccdak-schema-registry

nano src/main/java/com/linuxacademy/ccdak/schemaregistry/SchemaRegistryConsumerMain.java

Implement a consumer that deserializes data using an avro schema. This class uses the Person schema created in an earlier lesson at src/main/avro/com/linuxacademy/ccdak/schemaregistry/Person.avsc.

package com.linuxacademy.ccdak.schemaregistry;

import io.confluent.kafka.serializers.AbstractKafkaAvroSerDeConfig;

import io.confluent.kafka.serializers.KafkaAvroDeserializer;

import io.confluent.kafka.serializers.KafkaAvroDeserializerConfig;

import java.time.Duration;

import java.util.Collections;

import java.util.Properties;

import org.apache.kafka.clients.consumer.ConsumerConfig;

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.common.serialization.StringDeserializer;

public class SchemaRegistryConsumerMain {

public static void main(String[] args) {

final Properties props = new Properties();

props.put(ConsumerConfig.BOOTSTRAP\_SERVERS\_CONFIG, "localhost:9092");

props.put(ConsumerConfig.GROUP\_ID\_CONFIG, "group1");

props.put(ConsumerConfig.ENABLE\_AUTO\_COMMIT\_CONFIG, "true");

props.put(ConsumerConfig.AUTO\_COMMIT\_INTERVAL\_MS\_CONFIG, "1000");

props.put(ConsumerConfig.AUTO\_OFFSET\_RESET\_CONFIG, "earliest");

props.put(AbstractKafkaAvroSerDeConfig.SCHEMA\_REGISTRY\_URL\_CONFIG, "http://localhost:8081");

props.put(ConsumerConfig.KEY\_DESERIALIZER\_CLASS\_CONFIG, StringDeserializer.class);

props.put(ConsumerConfig.VALUE\_DESERIALIZER\_CLASS\_CONFIG, KafkaAvroDeserializer.class);

props.put(KafkaAvroDeserializerConfig.SPECIFIC\_AVRO\_READER\_CONFIG, true);

KafkaConsumer<String, Person> consumer = new KafkaConsumer<>(props);

consumer.subscribe(Collections.singletonList("employees"));

while (true) {

final ConsumerRecords<String, Person> records = consumer.poll(Duration.ofMillis(100));

for (final ConsumerRecord<String, Person> record : records) {

final String key = record.key();

final Person value = record.value();

System.out.println("key=" + key + ", value=" + value);

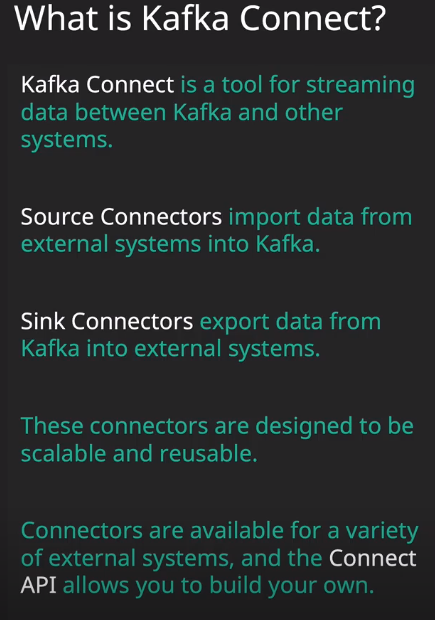
}

} }}

Run your code.

./gradlew runConsumer

=========================Kafka Connect ====================



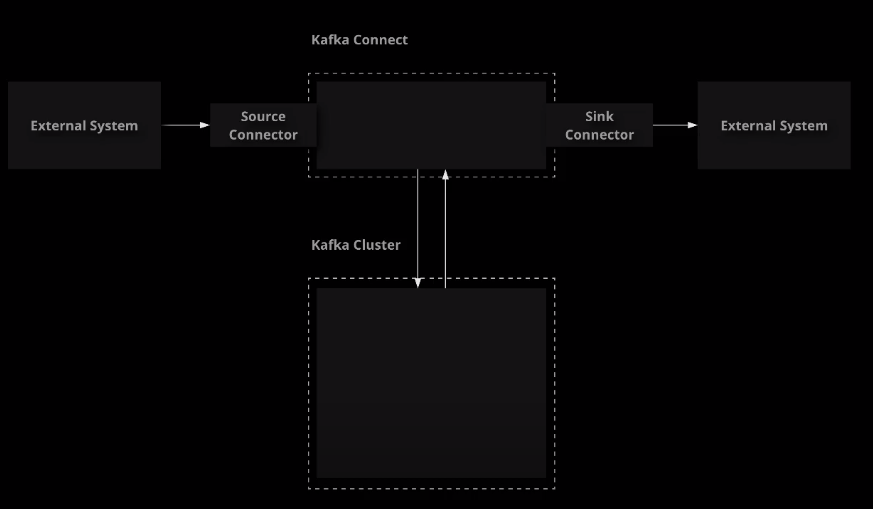
We have already explored how to get data into and out of Kafka using producers and consumers, but Kafka offers another method of moving data that is more tailored toward integrations with external systems: Kafka Connect.

Start and enable the Kafka Connect service on your first broker.

sudo systemctl start confluent-kafka-connect

sudo systemctl enable confluent-kafka-connect

sudo systemctl status confluent-kafka-connect



## **Using Kafka Connect**

Kafka Connect provides a useful framework for integrations between Kafka and external systems . We will demonstrate the process of configuring a simple source and sink connector. This will give you an idea of what it looks like to use Kafka Connect in practice.

Create a topic to use for testing.

kafka-topics --bootstrap-server localhost:9092 --create --topic connect\_topic --partitions 1 --replication-factor 1

Create the input and output files.

cd ~/

touch input.txt

touch output.txt

chmod 777 output.txt

Enter some data into the input file.

nano input.txt

Create a source connector to import data into Kafka from a file.

curl -X POST http://localhost:8083/connectors \

-H 'Accept: \*/\*' \

-H 'Content-Type: application/json' \

-d '{

"name": "file\_source\_connector",

"config": {

"connector.class": "org.apache.kafka.connect.file.FileStreamSourceConnector",

"topic": "connect\_topic",

"file": "/home/ubuntu/input.txt",

"value.converter": "org.apache.kafka.connect.storage.StringConverter"

}

}'

Get information about the source connector.

curl http://localhost:8083/connectors/file\_source\_connector

curl http://localhost:8083/connectors/file\_source\_connector/status

Check the topic to verify the new data has appeared.

kafka-console-consumer --bootstrap-server localhost:9092 --topic connect\_topic --from-beginning

Create a sink connector to export data from Kafka to a file.

curl -X POST http://localhost:8083/connectors \

-H 'Accept: \*/\*' \

-H 'Content-Type: application/json' \

-d '{

"name": "file\_sink\_connector",

"config": {

"connector.class": "org.apache.kafka.connect.file.FileStreamSinkConnector",

"topics": "connect\_topic",

"file": "/home/ubuntu/output.txt",

"value.converter": "org.apache.kafka.connect.storage.StringConverter"

}

}'

Check the contents of the output file.

cat /home/cloud\_user/output.txt

Delete both connectors to clean up.

curl -X DELETE http://localhost:8083/connectors/file\_source\_connector

curl -X DELETE http://localhost:8083/connectors/file\_sink\_connector