Confluent Certified Developer for Apache Kafka (CCDAK)

Study Guide

William Boyd
willliamb@linuxacademy.com
October 2019

Contents

Application Design **4**

Kafka Architecture Basics 4

Kafka and Java 9

Kafka Streams 10

Advanced Application Design Concepts 19

Development **23**

Working with Kafka in Java 23

Working with the Confluent Kafka REST APIs 27

Confluent Schema Registry 30

Kafka Connect 37

Deployment, Testing, and Monitoring **40**

Kafka Security 40

Testing 43

Working with Clients 51

Confluent KSQL 54

Application Design

Kafka Architecture Basics

What Is Apache Kafka?

Documentation

- **https://kafka.apache.org/documentation/#introduction**
- **https://kafka.apache.org/uses**

Apache Kafka: A distributed messaging and data streaming platform.

Background

- Written in Java and Scala
- Created at LinkedIn
- Open-sourced in 2011

Primary Use Cases

- **Messaging:** Reliably passing data between systems, with fault tolerance and without data loss

- **Streaming:** Building applications that process and transform streams of data in real time

Kafka offers:

- Strong reliability guarantees
- Fault tolerance
- Robust APIs

Kafka from the Command Line

Documentation

- **https://kafka.apache.org/documentation/#quickstart**
- **https://docs.confluent.io/current/cli/index.html**

Kafka Shell Scripts

- Installed in the `bin/` under your Kafka installation directory (e.g., `bin/kafka-topics.sh`)
- With Confluent, they are placed in `/usr/bin`, so you can reference these scripts like: `kafka-topics`.

The `confluent` command allows you to manage certain Confluent-specific features in a non-production environment.

List topics on a Kafka-only install:

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

List topics on a Confluent package install:

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

Publisher/Subscriber Messaging in Kafka

Documentation

- **https://kafka.apache.org/documentation/#intro_topics**

Important Concepts

Publisher/Subscriber Messaging (pub/sub): Applications write data to the messaging system (publishers), and other applications read the data (subscribers).

- **Topic:** A named data feed that can be written to and read from
- **Log:** The underlying data structure used to store a topic's data. It is a partitioned, immutable sequence of data records.
- **Partition:** A section of a topic's log. Topic logs are broken up into partitions for scalability.
- **Offset:** The sequential, unique ID of a record in a partition
- **Producer:** An application that writes records to a topic
- **Consumer:** An application that reads records from a topic
- **Consumer group:** A collection of multiple consumers. A consumer group will collectively process each record in the topic exactly once, rather than each individual consumer processing all records.

Kafka Architecture

Documentation

- https://kafka.apache.org/documentation/#quickstart_multibroker
- https://cwiki.apache.org/confluence/display/KAFKA/Kafka+Controller+Internals

Important Concepts

- **Broker:** A Kafka server that is part of a cluster
- **Zookeeper:** Cluster management tool used to manage the Kafka cluster. Coordinates communication between brokers, monitors broker status, handles adding and removing brokers, etc.
- **Kafka Networking Protocol:** Kafka used the `TCP` protocol for cluster communication.
- **Controller:** A single broker in the cluster is automatically selected to be the controller. The controller coordinates the process of assigning partitions and replicas to individual brokers. The cluster will always have exactly one controller.

Partitions and Replication

Documentation

- https://kafka.apache.org/documentation/#replication

Important Concepts

- **Replica:** A copy of a partition. When there are multiple replicas of a partition, they must all reside on different brokers.
- **Replication factor:** A property of a Kafka topic. The replication factor determines how many copies there will be of each partition for that topic.

You can use the `--partitions` and `--replication-factor` parameters with `kafka-topics` to set the number of partitions and replication factor when creating a topic from the command line:

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

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

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

- **Leader:** A replica serves as a source of truth among multiple replicas. Producers write to and consumers read from the leader.
- **Leader election:** The process of selecting which replica will be the leader. If the leader becomes unavailable, a new round of leader election occurs.
- **In-sync replica (ISR):** A replica that is in sync with the leader (contains all of the data that the leader has). By default, only in-sync replicas can be chosen as leaders. If no ISRs are available, no data can be written to the partition, forcing producers to pause.
- **Unclean leader election:** A configuration setting that allows a non-in-sync replica to be elected as a leader if no ISRs are available

Kafka and Java

The Kafka Java APIs

Documentation

- **https://kafka.apache.org/documentation/#api**

Important Concepts

Kafka offers application programming interfaces (APIs), which make it easier to build applications that use Kafka.

The official Kafka API libraries are written for Java, but there are open-source APIs for other languages as well.

The Five Kafka APIs

- **Producer API:** Build applications that write data to Kafka topics.
- **Consumer API:** Build applications that read data from Kafka topics.
- **Streams API:** Build Kafka Streams applications, which read from input topics, transform data, and write to output topics.
- **Connect API:** Build custom connectors — applications that pass data between Kafka and specific external systems.
- **AdminClient API:** Programmatically manage administrative objects like topics and brokers.

An example of a basic producer using the Java Producer API. This producer counts to 100 and publishes each number in the count to a Kafka topic:

```
Properties props = new Properties();
props.put("bootstrap.servers", "localhost:9092");
props.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");
props.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer");
Producer<String, String> producer = new KafkaProducer<>(props);
for (int i = 0; i < 100; i++) {
    producer.send(new ProducerRecord<String, String>("count-topic", "count", Integer.toString(i)));
}
producer.close();
```

Kafka Streams

What Are Streams?

Documentation

- **https://kafka.apache.org/23/documentation/streams/**

Important Concepts

Kafka Streams: Feature set that allows you to build applications that process data in real time. A Streams application's input and output are usually Kafka topics.

Some basic stream code that reads from an input topic and writes to an output topic:

```
// Set up the configuration.
final Properties props = new Properties();
props.put(StreamsConfig.APPLICATION_ID_CONFIG, "inventory-data");
props.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
```

```
props.put(StreamsConfig.CACHE_MAX_BYTES_BUFFERING_CONFIG, 0);
// Since the input topic uses Strings for both key and value, set the default Serdes to String.
props.put(StreamsConfig.DEFAULT_KEY_SERDE_CLASS_CONFIG, Serdes.String().getClass().getName());
props.put(StreamsConfig.DEFAULT_VALUE_SERDE_CLASS_CONFIG, Serdes.String().getClass().getName());

// Get the source stream.
final StreamsBuilder builder = new StreamsBuilder();
final KStream<String, String> source = builder.stream("streams-input-topic");

source.to("streams-output-topic");
```

Kafka Streams Stateless Transformations

Documentation

- **https://kafka.apache.org/23/documentation/streams/developer-guide/dsl-api.html#stateless-transformations**

Important Concepts

Stateful transformations: Streams operations that require a special data store in the Kafka cluster to store the current state of the stream. When processing a record, they need information about other records that is stored in the state store.

Stateless transformations: Streams operations that do not require a special data store. These only need information about the current record being processed.

Some Stateless Transformations

- **Branch:** Split a stream into multiple streams, dividing up the records based on a condition.

```
// Split the stream into two streams, one containing all records where the key begins with "a",
and the other containing all other records.
KStream<String, String>[] branches = stream
    .branch((key, value) -> key.startsWith("a"), (key, value) -> true);
KStream<String, String> aKeysStream = branches[0];
KStream<String, String> othersStream = branches[1];
```

- **Filter:** Remove records from the stream based on a condition.

```
// Remove any records from the stream where the value does not start with "a".
KStream<String, String> filteredStream = stream.filter((key, value) -> value.startsWith("a"));
```

- **FlatMap:** Convert input records into any number of output records.

```
// Convert each record into two records, one with an uppercased value and one with a lowercased
value.
KStream<String, String> flatMapStream = stream.flatMap((key, value) -> {
    List<KeyValue<String, String>> result = new LinkedList<>();
    result.add(KeyValue.pair(key, value.toUpperCase()));
    result.add(KeyValue.pair(key, value.toLowerCase()));
    return result;
});
```

- **Foreach:** Perform some arbitrary stateless logic on each record. This is a terminal operation that prevents any further processing of the stream, including writing to an output topic.

```
//Print each record to the console.
stream.foreach((key, value) -> System.out.println("key=" + key + ", value=" + value));
```

- **GroupBy:** Group the stream by a new key determined programmatically.

```
// Group the stream by a new key which is a lowercased version of the old key.
KGroupedStream<String, String> groupedStream = stream.groupBy(
  (key, value) -> key.toLowerCase()
);
```

- **GroupByKey:** Group the stream by an existing key.

```
KGroupedStream<String, String> groupedStream = stream.groupByKey();
```

- **Map:** Convert each record into exactly one new, modified record.

```
// Modify all records by uppercasing the key.
KStream<String, String> mapStream = stream.map((key, value) -> KeyValue.pair(key.toUpperCase(),
value));
```

- **Merge:** Combine two streams into one stream. The new stream will contain all records from both streams.

```
//Merge two streams together.
KStream<String, String> mergedStream = stream.merge(anotherStream);
```

- **Peek:** Perform a stateless, arbitrary operation on each record. Peek is a non-terminal operation and allows further processing.

```
//Print each record to the console.
KStream<String, String> peekedStream = stream.peek((key, value) -> System.out.println("key=" +
key + ", value=" + value));
```

Kafka Streams Aggregations

Documentation

- https://kafka.apache.org/23/documentation/streams/developer-guide/dsl-api.html#aggregating

Important Concepts

Aggregation: A stateful transformation performed across a set of records grouped by their key (using GroupBy or GroupByKey).

Kafka Aggregations

- **Aggregate:** A generalized aggregation, generates a new record based on a calculation involving the grouped input records.

```
// Create an aggregation that totals the length in characters of the value for all records
sharing the same key.
KTable<String, Integer> aggregatedTable = groupedStream.aggregate(
    () -> 0,
    (aggKey, newValue, aggValue) -> aggValue + newValue.length());
aggregatedTable.toStream().to("aggregations-output-charactercount-topic"));
```

- **Count:** Counts the number of records in a group.

```
// Count the number of records for each key.
KTable<String, Long> countedTable = groupedStream.count();
countedTable.toStream().to("aggregations-output-count-topic");
```

- **Reduce:** Combines grouped records into a single record with a value calculated from the values of of the input records.

```
// Combine the values of all records with the same key into a string separated by spaces.
KTable<String, String> reducedTable = groupedStream.reduce((aggValue, newValue) -> aggValue + "
" + newValue);
reducedTable.toStream().to("aggregations-output-reduce-topic");
```

Kafka Streams Joins

Documentation

- **https://kafka.apache.org/23/documentation/streams/developer-guide/dsl-api.html#joining**

Important Concepts

Joins: Stateful transformations that combine two input streams. Records that share the same key in the input streams are merged into a single record. Joins in Kafka are similar to SQL joins.

Co-partitioning: Topics must be co-partitioned in order to perform joins (unless using a GlobalKTable). * The topics must have the same number of partitions. * They must use the same partitioning/keying strategy (i.e., the same keys must be in analogous partitions across both topics).

GlobalKTable: A KTable that maintains a copy of all partitions for every instance of the Streams application. When using GlobalKTables, you can ignore co-partitioning requirements when performing joins.

Join Types

- **Inner join:** The new stream will only contain records that exist in both input streams.

```
// Perform an inner join.
KStream<String, String> innerJoined = left.join(
    right,
    (leftValue, rightValue) -> "left=" + leftValue + ", right=" + rightValue,
    JoinWindows.of(Duration.ofMinutes(5)));
innerJoined.to("inner-join-output-topic");
```

- **Left join:** The new stream will contain all records from the "left" stream, and will combine them with the matching records from the "right" stream, if they exist.

```
// Perform a left join.
KStream<String, String> innerJoined = left.leftJoin(
    right,
    (leftValue, rightValue) -> "left=" + leftValue + ", right=" + rightValue,
    JoinWindows.of(Duration.ofMinutes(5)));
innerJoined.to("inner-join-output-topic");
```

- **Outer join:** The new stream will contain all records from both streams. Records that exist in both streams will be combined.

```
// Perform an outer join.
KStream<String, String> innerJoined = left.outerJoin(
    right,
    (leftValue, rightValue) -> "left=" + leftValue + ", right=" + rightValue,
    JoinWindows.of(Duration.ofMinutes(5)));
innerJoined.to("inner-join-output-topic");
```

Kafka Streams Windowing

Documentation

- **https://kafka.apache.org/23/documentation/streams/developer-guide/dsl-api.html#windowing**

Important Concepts

Windows: Windows subdivide groups of records into time buckets.

Window Types

- **Tumbling time windows:** Time periods with no overlap or gaps between them
- **Hopping time windows:** Time periods with overlaps and/or gaps between them
- **Sliding time windows:** Used only for joins. Sliding time windows are defined by the length of time between the timestamps of any two records rather than a specific point in time.
- **Session windows:** Dynamic windows formed around periods of "activity" and "idle time" based on the timestamps of input records. There will be no window during times when there are no records.

A basic aggregation using a tumbling time window:

Think “What is the status of the last 10 minutes?”. Good exam
The number of visitors in the last 10 minutes, updated every r
The most viewed products in the last 10 minutes, continously u

```
KGroupedStream<String, String> groupedStream = source.groupByKey();

// Apply windowing to the stream with tumbling time windows of 10 seconds.
TimeWindowedKStream<String, String> windowedStream =
groupedStream.windowedBy(TimeWindows.of(Duration.ofSeconds(10)));

// Combine the values of all records with the same key into a string separated by spaces, using 10-
```

```
second windows.
KTable<Windowed<String>, String> reducedTable = windowedStream.reduce((aggValue, newValue) ->
aggValue + " " + newValue);
reducedTable.toStream().to("windowing-output-topic",
Produced.with(WindowedSerdes.timeWindowedSerdeFrom(String.class), Serdes.String()));
```

Late-arriving records: Records that arrive after their window has passed. Windows are kept in storage for a period of time to allow processing of late-arriving records.

Window retention period: The configurable period of time to maintain window data in storage after a window has passed. Late-arriving records can still be processed during this time, but will be ignored after the retention period has passed.

Streams vs. Tables

Kafka Streams can model data as either a **stream** or **table**.

- **Stream:** Each record is a self-contained piece of data in an unbounded set of data. New records do not replace an existing piece of data with a new value.
- **Table:** Records represent a current state that can be overwritten/updated.

Example Use Cases

Stream

- Credit card transactions in real time
- A real-time log of attendees checking in to a conference
- A log of customer purchases that represent the removal of items from a store's inventory

Table

- A user's current available credit card balance
- A list of conference attendee names with a value indicating whether or not they have checked in
- A set of data containing the quantity of each item in a store's inventory

Advanced Application Design Concepts

Kafka Configuration

Documentation

- **https://kafka.apache.org/documentation/#configuration**
- **https://kafka.apache.org/documentation/#config**
- **https://kafka.apache.org/documentation/#brokerconfigs**
- **https://kafka.apache.org/documentation/#topicconfigs**

Important Concepts

Broker Configs

Modify `server.properties` or use `kafka-configs` to interact with broker configuration.

List the current configurations for broker 1:

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

Modify the configuration for broker 1 by setting `log.cleaner.threads=2`:

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

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 Configs

You can specify configurations for a new topic when creating it using `kafka-topics`:

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

You can also interact with the configs for existing topics using `kafka-configs` with `--entity-type topics`.

List the configurations for the topic to see the configuration override:

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

Modify a 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
```

List the topic configurations again to see the changes:

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

Topic configurations can have broker-wide defaults that will apply to topics that don't have a per-topic override for that configuration.

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
```

View the broker configuration to see the changes to the default:

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

Client Configs

You can configure clients programmatically in Java using a `Properties` object. For example:

```
Properties props = new Properties();
props.put("bootstrap.servers", "localhost:9092");
props.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");
props.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer");
Producer<String, String> producer = new KafkaProducer<>(props);
```

Topic Design

Number of partitions and replication factor are the two main components of designing a topic.

Some questions to consider when designing a topics:

- How many brokers do you have?
  - The number of brokers limits the number of replicas.
- What is your need for fault tolerance?
  - A higher replication factor means greater fault tolerance.

- How many consumers do you want to place in a consumer group for parallel processing?
  - You will need at least as many partitions as the number of consumers you expect to have on a single group.
- How much memory is available on each broker?
  - Kafka requires memory to process messages. The configuration setting replica.fetch.max.bytes (default ~1 MB) determines the rough amount of memory you will need for each partition on a broker.

Metrics and Monitoring

Documentation

- **https://kafka.apache.org/documentation/#monitoring**

Important Concepts

Kafka exposes monitoring data via JMX.

To access Kafka metrics, enable JMX connections appropriately for your use case. You can pass in JMX options via the `KAFKA_JMX_OPTS` environment variable:

```
KAFKA_JMX_OPTS="-Dcom.sun.management.jmxremote -Dcom.sun.management.jmxremote.local.only=false -
Djava.rmi.server.hostname=localhost"
```

You collect to JMX and collect metrics using any JMX client.

Development

Working with Kafka in Java

Building a Producer in Java

Documentation

- https://kafka.apache.org/documentation/#producerapi
- https://kafka.apache.org/23/javadoc/index.html?org/apache/kafka/clients/producer/KafkaProducer.html

Important Concepts

You can publish data to Kafka in your Java code using Kafka's Java Producer API. The `KafkaProducer` class handles interaction between your code and Kafka.

Here is an example of a Java producer running inside a `Main` class. This producer counts from 0 to 99 and publishes each count to a topic:

```
package com.linuxacademy.ccdak.clients;

import java.util.Properties;
import org.apache.kafka.clients.producer.KafkaProducer;
import org.apache.kafka.clients.producer.Producer;
import org.apache.kafka.clients.producer.ProducerRecord;
```

```
import org.apache.kafka.clients.producer.RecordMetadata;

public class ProducerMain {

    public static void main(String[] args) {
        Properties props = new Properties();
        props.put("bootstrap.servers", "localhost:9092");
        props.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");
        props.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer");

        props.put("acks", "all");

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

        for (int i = 0; i < 100; i++) {
            int partition = 0;
            if (i > 49) {
                partition = 1;
            }
            ProducerRecord record = new ProducerRecord<>("test_count", partition, "count",
Integer.toString(i));
            producer.send(record, (RecordMetadata metadata, Exception e) -> {
                if (e != null) {
                    System.out.println("Error publishing message: " + e.getMessage());
                } else {
                    System.out.println("Published message: key=" + record.key() +
                            ", value=" + record.value() +
                            ", topic=" + metadata.topic() +
                            ", partition=" + metadata.partition() +
                            ", offset=" + metadata.offset());
                }
```

```
            });
        }

        producer.close();
    }

}
```

Building a Consumer in Java

Documentation

- **https://kafka.apache.org/documentation/#consumerapi**
- **https://kafka.apache.org/23/javadoc/index.html?org/apache/kafka/clients/consumer/KafkaConsumer.html**

Important Concepts

You can build your own consumers using the Java Consumer API. The `KafkaConsumer` class handles interactions between your code and Kafka.

Here is an example of a consumer. This consumer reads records from two topics and outputs the data to the console:

```
package com.linuxacademy.ccdak.clients;

import java.time.Duration;
import java.util.Arrays;
import java.util.Properties;
import org.apache.kafka.clients.consumer.ConsumerRecord;
```

```
import org.apache.kafka.clients.consumer.ConsumerRecords;
import org.apache.kafka.clients.consumer.KafkaConsumer;

public class ConsumerMain {

    public static void main(String[] args) {
        Properties props = new Properties();
        props.setProperty("bootstrap.servers", "localhost:9092");
        props.setProperty("group.id", "group1");
        props.setProperty("enable.auto.commit", "false");
        props.setProperty("key.deserializer",
"org.apache.kafka.common.serialization.StringDeserializer");
        props.setProperty("value.deserializer",
"org.apache.kafka.common.serialization.StringDeserializer");
        KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props);
        consumer.subscribe(Arrays.asList("test_topic1", "test_topic2"));
        while (true) {
            ConsumerRecords<String, String> records = consumer.poll(Duration.ofMillis(100));
            for (ConsumerRecord<String, String> record : records) {
                System.out.println("key=" + record.key() + ", value=" + record.value() + ", topic="
+ record.topic() + ", partition=" + record.partition() + ", offset=" + record.offset());
            }
            consumer.commitSync();
        }
    }

}
```

Working with the Confluent Kafka REST APIs

The Confluent REST Proxy

Documentation

- **https://docs.confluent.io/current/kafka-rest/index.html**
- **https://docs.confluent.io/current/kafka-rest/api.html**

Important Concepts

Confluent REST Proxy provides a RESTful interface for Kafka.

It runs as a separate service that acts as a middleman between REST clients and the Kafka cluster.

With Confluent REST Proxy, you can do things like produce and consume messages using HTTP requests.

Producing Messages with REST Proxy

Documentation

- **https://docs.confluent.io/current/kafka-rest/quickstart.html**
- **https://docs.confluent.io/current/kafka-rest/api.html**

Important Concepts

You can publish records to a topic with a simple POST request:

```
POST /topics/<topic_name>

{
  "records":[
    {
      "key": "<key>",
      "value": "<value>"
    },
    {
      "key": "<key>",
      "value": "<value>"
    }
  ]
}
```

Consuming Messages with REST Proxy

Documentation

- **https://docs.confluent.io/current/kafka-rest/quickstart.html**
- **https://docs.confluent.io/current/kafka-rest/api.html**

Important Concepts

To consume messages using Confluent REST Proxy, you must first create a consumer and consumer instance:

```
POST /consumers/<consumer_name>

{
  "name":"<consumer instance name>",
  "format":"json",
  "auto.offset.reset":"earliest"
}
```

Then, subscribe your consumer to a topic:

```
POST /consumers/<consumer name>/instances/<consumer instance name>/subscription

{
  "topics":[
    "<topic name>"
  ]
}
```

Consume messages using the consumer:

```
GET /consumers/<consumer name>/instances/<consumer instance name>/records
```

You can also delete consumer instances if you no longer need them:

```
DELETE /consumers/<consumer name>/instances/<consumer instance name>
```

Confluent Schema Registry

What Is Confluent Schema Registry?

Documentation

- **https://docs.confluent.io/current/schema-registry/index.html**

Important Concepts

- **Schema:** Metadata that describes a complex data format, including what fields are expected and their data types. Schemas can be used to define complex data type with multiple typed fields for both keys and values in Kafka records.
- **Apache Avro:** Apache technology for defining and using schemas.
- **Confluent Schema Registry:** A central location to store and access Avro schemas.

Producers can register a schema with Schema Registry, and then use that schema to serialize data. Consumers can then download the schema from Schema Registry and use it to deserialize the data.

Creating an Avro Schema

Documentation

- **https://docs.confluent.io/current/schema-registry/schema_registry_tutorial.html**
- **https://avro.apache.org/docs/current/spec.html#schemas**

Important Concepts

Avro schemas can be defined using JSON:

```
{
  "namespace":"<namespace>",
  "type": "record",
  "name": "<schema name>",
  "fields": [
    {
      "name": "<field name>",
      "type": "<field type>"
    }
  ]
}
```

Here is an example schema 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

Documentation

- **https://docs.confluent.io/current/schema-registry/schema_registry_tutorial.html**
- **https://github.com/confluentinc/examples/blob/5.3.0-post/clients/avro/src/main/java/io/confluent/examples/clients/basicavro/ProducerExample.java**

Important Concepts

To use Schema Registry with a Kafka producer, include a schema definition file in your project. With Gradle, you can use the Avro Gradle plugin to help locate the schema file and automatically generate a Java class from it.

In `build.gradle`:

```
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'
}

...
```

Here is an example of a producer that uses Schema Registry. Note the configuration properties and the use of the `Person` class, a class automatically generated from a schema by the Avro plugin:

```
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();
    }

}
```

Using Schema Registry with a Kafka Consumer

Documentation

- **https://docs.confluent.io/current/schema-registry/schema_registry_tutorial.html**
- **https://github.com/confluentinc/examples/blob/5.3.0-post/clients/avro/src/main/java/io/confluent/examples/clients/basicavro/ConsumerExample.java**

Important Concepts

You can use Schema Registry in your consumers as well.

Here is an example of a consumer using schema registry. Note the schema-related configuration properties and the use of the `Person` class, a class generated from the schema file by the Avro plugin:

```
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);
            }
        }
    }

}
```

Managing Changes to an Avro Schema

Documentation

- **https://docs.confluent.io/current/schema-registry/avro.html**
- **https://docs.confluent.io/current/schema-registry/schema_registry_tutorial.html#schema-evolution-and-compatibility**

Important Concepts

Schema Registry has a schema compatibility checking feature. This allows you to set a contract that determines what changes can and cannot be made to a schema. If you attempt to register a schema change that breaks this contract, you will get an error. This helps you enforce your change management strategy for schemas.

The contract used is determined by a schema's compatibility type. The following compatibility types are available:

- **BACKWARD:** (Default) Consumers using an updated schema can read data that was serialized via the current schema in the registry.
- **BACKWARD_TRANSITIVE:** A consumer using an updated schema can use data serialized via all previously registered schemas.
- **FORWARD:** Consumers using the current schema in the registry can read data serialized via the updated schema.
- **FORWARD_TRANSITIVE:** Consumers using any previous schema in the registry can read data serialized via the updated schema.
- **FULL:** The new schema is forward- and backward-compatible with the current schema in the registry.
- **FULL_TRANSITIVE:** The new schema is forward- and backward-compatible with all previous schemas in the registry.
- **NONE:** All compatibility checks are disabled.

Kafka Connect

What Is Kafka Connect?

Documentation

- **https://kafka.apache.org/documentation/#connect**

Important Concepts

Kafka Connect is a tool that helps you integrate Kafka with other systems. It allows you to stream data between Kafka and those other systems in real time.

- **Source connectors:** Import data from an external system into Kafka.
- **Sink connectors:** Export data from Kafka to an external system.

Using Kafka Connect

Documentation

- **https://kafka.apache.org/documentation.html#connect**
- **https://docs.confluent.io/current/connect/filestream_connector.html**

You can create and manage connectors through a RESTful API. The `connector.class` field specifies a connector implementation that determines how the connector behaves and what external system(s) it integrates with. Confluent includes a variety of pre-made connector classes for common integration use cases.

An example of how to create a source connector that reads data from a file and imports it into Kafka:

```
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/cloud_user/input.txt",
        "value.converter": "org.apache.kafka.connect.storage.StringConverter"
    }
}'
```

You can get information about the source connector like so:

```
curl http://localhost:8083/connectors/file_source_connector

curl http://localhost:8083/connectors/file_source_connector/status
```

An example of how to 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/cloud_user/output.txt",
        "value.converter": "org.apache.kafka.connect.storage.StringConverter"
    }
}'
```

Deployment, Testing, and Monitoring

Kafka Security

TLS Encryption

Documentation

- **http://kafka.apache.org/documentation.html#security_ssl**

Important Concepts

You can enable TLS encryption to allow encryption of traffic between clients and Kafka and defend against man-in-the-middle attacks.

You will need to obtain or generate certificate authority and certificate files for your Brokers.

In `server.properties`, add an `SSL` listener to listen on a TLS-enabled port. Then, add `ssl.` configurations to tell the broker how to locate and access the certificate files it needs to use. The final config could look something like this:

```
listeners=PLAINTEXT://<hostname>:9092,SSL://<hostname>:9093

ssl.keystore.location=/var/private/ssl/server.keystore.jks
ssl.keystore.password=<keystore password>
ssl.key.password=<broker key password>
ssl.truststore.location=/var/private/ssl/server.truststore.jks
```

```
ssl.truststore.password=<trust store password>
ssl.client.auth=none
```

Client Authentication

Documentation

- **http://kafka.apache.org/documentation.html#security_ssl**

Important Concepts

You can enable client authentication for greater security by allowing (or even requiring) clients to authenticate. There are multiple ways to handle client authentication, including the use of client certificates.

To authenticate using client certificates, you will need a client certificate for the client signed using the certificate authority.

Enable SSL client authentication in `server.properties`:

```
ssl.client.auth=required
```

You can configure a client to use a client certificate like so:

```
ssl.keystore.location=/var/private/ssl/client.keystore.jks
ssl.keystore.password=<client keystore password>
ssl.key.password=<client key password>
```

ACL Authorization

Documentation

- **https://kafka.apache.org/documentation/#security_authz**

Important Concepts

Kafka uses access control lists (ACLs) to manage authorization.

An ACL consists of the following components:

- **Principal:** The user
- **Allow/Deny
- **Operation:** What the user can do (e.g., read or write)
- **Host:** The IP address(es) of hosts connecting to the cluster to perform this action
- **Resource pattern:** A pattern that matches one or more resources, such as topics

In `server.properties`, enable ACL authorization. Note that the `ssl.principal.mapping.rules` configuration is specific to SSL client authentication:

```
authorizer.class.name=kafka.security.auth.SimpleAclAuthorizer
super.users=User:admin
allow.everyone.if.no.acl.found=true
ssl.principal.mapping.rules=RULE:^CN=(.*?),OU=.*$/$1/,DEFAULT
```

You can manage ACLs using the `kafka-acls` command. For example, this command will add an ACL to allow `myuser` to write to the `acl-test` topic:

```
kafka-acls --authorizer-properties zookeeper.connect=localhost:2181 --add --allow-principal
User:myuser --operation write --topic acl-test
```

Testing

Testing Producers

Documentation

- **https://kafka.apache.org/20/javadoc/org/apache/kafka/clients/producer/MockProducer.html**

Important Concepts

The Kafka Producer API provides a `MockProducer` class. You can use `MockProducer` to simulate the behavior of a real `KafkaProducer` in your unit tests.

Here is an example of a unit test class using `MockProducer`. You can find the full project, including the code being tested, at **https://github.com/linuxacademy/content-ccdak-testing/tree/end-state**.

```
package com.linuxacademy.ccdak.testing;

import java.io.ByteArrayOutputStream;
import java.io.PrintStream;
import java.util.List;
import org.apache.kafka.clients.producer.MockProducer;
import org.apache.kafka.clients.producer.ProducerRecord;
import org.apache.kafka.common.serialization.IntegerSerializer;
import org.apache.kafka.common.serialization.StringSerializer;
```

```
import org.junit.After;
import org.junit.Assert;
import org.junit.Before;
import org.junit.Test;

public class MyProducerTest {

    MockProducer<Integer, String> mockProducer;
    MyProducer myProducer;

    // Contains data sent so System.out during the test.
    private ByteArrayOutputStream systemOutContent;
    // Contains data sent so System.err during the test.
    private ByteArrayOutputStream systemErrContent;
    private final PrintStream originalSystemOut = System.out;
    private final PrintStream originalSystemErr = System.err;

    @Before
    public void setUp() {
        mockProducer = new MockProducer<>(false, new IntegerSerializer(), new StringSerializer());
        myProducer = new MyProducer();
        myProducer.producer = mockProducer;
    }

    @Before
    public void setUpStreams() {
        systemOutContent = new ByteArrayOutputStream();
        systemErrContent = new ByteArrayOutputStream();
        System.setOut(new PrintStream(systemOutContent));
        System.setErr(new PrintStream(systemErrContent));
    }
```

```
    @After
    public void restoreStreams() {
        System.setOut(originalSystemOut);
        System.setErr(originalSystemErr);
    }

    @Test
    public void testPublishRecord_sent_data() {
        // Perform a simple test to verify that the producer sends the correct data to the correct
topic when publishRecord is called.
        myProducer.publishRecord(1, "Test Data");

        mockProducer.completeNext();

        List<ProducerRecord<Integer, String>> records = mockProducer.history();
        Assert.assertEquals(1, records.size());
        ProducerRecord<Integer, String> record = records.get(0);
        Assert.assertEquals(Integer.valueOf(1), record.key());
        Assert.assertEquals("Test Data", record.value());
        Assert.assertEquals("test_topic", record.topic());
        Assert.assertEquals("key=1, value=Test Data\n", systemOutContent.toString());
    }

}
```

Testing Consumers

Documentation

- **https://kafka.apache.org/20/javadoc/org/apache/kafka/clients/consumer/MockConsumer.html**

Important Concepts

The `MockConsumer` class can help you build unit tests for your Kafka consumers.

Here is an example of a unit test using `MockConsumer`. You can find the full project, including the code being tested, at **https://github.com/linuxacademy/content-ccdak-testing/tree/end-state**.

```
package com.linuxacademy.ccdak.testing;

import java.io.ByteArrayOutputStream;
import java.io.PrintStream;
import java.util.Arrays;
import java.util.HashMap;
import org.apache.kafka.clients.consumer.ConsumerRecord;
import org.apache.kafka.clients.consumer.MockConsumer;
import org.apache.kafka.clients.consumer.OffsetResetStrategy;
import org.apache.kafka.common.TopicPartition;
import org.junit.After;
import org.junit.Assert;
import org.junit.Before;
import org.junit.Test;

/**
```

```
 *
 * @author will
 */
public class MyConsumerTest {

    MockConsumer<Integer, String> mockConsumer;
    MyConsumer myConsumer;

    // Contains data sent so System.out during the test.
    private ByteArrayOutputStream systemOutContent;
    private final PrintStream originalSystemOut = System.out;

    @Before
    public void setUp() {
        mockConsumer = new MockConsumer<>(OffsetResetStrategy.EARLIEST);
        myConsumer = new MyConsumer();
        myConsumer.consumer = mockConsumer;
    }

    @Before
    public void setUpStreams() {
        systemOutContent = new ByteArrayOutputStream();
        System.setOut(new PrintStream(systemOutContent));
    }

    @After
    public void restoreStreams() {
        System.setOut(originalSystemOut);
    }

    @Test
```

```
    public void testHandleRecords_output() {
        // Verify that the testHandleRecords writes the correct data to System.out
        String topic = "test_topic";
        ConsumerRecord<Integer, String> record = new ConsumerRecord<>(topic, 0, 1, 2, "Test value");

        mockConsumer.assign(Arrays.asList(new TopicPartition(topic, 0)));
        HashMap<TopicPartition, Long> beginningOffsets = new HashMap<>();
        beginningOffsets.put(new TopicPartition("test_topic", 0), 0L);
        mockConsumer.updateBeginningOffsets(beginningOffsets);

        mockConsumer.addRecord(record);

        myConsumer.handleRecords();
        Assert.assertEquals("key=2, value=Test value, topic=test_topic, partition=0, offset=1\n",
systemOutContent.toString());
    }

}
```

Testing Streams Applications

Documentation

- **https://kafka.apache.org/11/documentation/streams/developer-guide/testing.html**

Important Concepts

Kafka provides a library called `kafka-streams-test-utils` that contains test fixtures for Kafka Streams applications, such as:

- **TopologyTestDriver:** Allows you to feed test records in, simulates your topology, and returns output records.
- **ConsumerRecordFactory:** Helps you convert consumer record data into byte arrays that can be processed by the TopologyTestDriver.
- **OutputVerifier:** Provides helper methods for verifying output records in your tests.

Here is an example of a unit test for a Streams application. You can find the full project, including the code being tested, at **https://github.com/linuxacademy/content-ccdak-testing/tree/end-state**.

```
package com.linuxacademy.ccdak.testing;

import java.util.Properties;
import org.apache.kafka.clients.consumer.ConsumerRecord;
import org.apache.kafka.clients.producer.ProducerRecord;
import org.apache.kafka.common.serialization.IntegerDeserializer;
import org.apache.kafka.common.serialization.IntegerSerializer;
import org.apache.kafka.common.serialization.Serdes;
import org.apache.kafka.common.serialization.StringDeserializer;
import org.apache.kafka.common.serialization.StringSerializer;
import org.apache.kafka.streams.StreamsConfig;
import org.apache.kafka.streams.Topology;
import org.apache.kafka.streams.TopologyTestDriver;
import org.apache.kafka.streams.test.ConsumerRecordFactory;
import org.apache.kafka.streams.test.OutputVerifier;
import org.junit.After;
import org.junit.Before;
```

```
import org.junit.Test;

public class MyStreamsTest {

    MyStreams myStreams;
    TopologyTestDriver testDriver;

    @Before
    public void setUp() {
        myStreams = new MyStreams();
        Topology topology = myStreams.topology;

        Properties props = new Properties();
        props.put(StreamsConfig.APPLICATION_ID_CONFIG, "test");
        props.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, "dummy:1234");
        props.put(StreamsConfig.DEFAULT_KEY_SERDE_CLASS_CONFIG,
Serdes.Integer().getClass().getName());
        props.put(StreamsConfig.DEFAULT_VALUE_SERDE_CLASS_CONFIG,
Serdes.String().getClass().getName());
        testDriver = new TopologyTestDriver(topology, props);
    }

    @After
    public void tearDown() {
        testDriver.close();
    }

    @Test
    public void test_first_name() {
        // Verify that the stream reverses the record value.
        ConsumerRecordFactory<Integer, String> factory = new
```

```
ConsumerRecordFactory<>("test_input_topic", new IntegerSerializer(), new StringSerializer());
        ConsumerRecord<byte[], byte[]> record = factory.create("test_input_topic", 1, "reverse");
        testDriver.pipeInput(record);

        ProducerRecord<Integer, String> outputRecord = testDriver.readOutput("test_output_topic",
new IntegerDeserializer(), new StringDeserializer());

        OutputVerifier.compareKeyValue(outputRecord, 1, "esrever");
    }

}
```

Working with Clients

Monitoring Clients

Documentation

- **https://kafka.apache.org/documentation/#producer_monitoring**
- **https://kafka.apache.org/documentation/#consumer_monitoring**

Important Concepts

Just like with brokers, you can monitor Kafka clients using JMX.

Some Important Producer Metrics

- **response-rate (global and per broker):** Responses (acks) received per second. Sudden changes in this value could signal a problem, though what the problem could be depends on your configuration.
- **request-rate (global and per broker):** Average requests sent per second. Requests can contain multiple records, so this is not the number of records. It does give you part of the overall throughput picture.
- **request-latency-avg (per broker):** Average request latency in ms. High latency could be a sign of performance issues, or just large batches. How long a particular request takes in order to complete and receive a response.
- **outgoing-byte-rate (global and per broker):** Bytes sent per second. Good picture of your network throughput. Helps with network planning.
- **io-wait-time-ns-avg (global only):** Average time spent waiting for a socket ready for reads/writes in nanoseconds. High wait times might mean your producers are producing more data than the cluster can accept and process.

Some Important Consumer Metrics

- **records-lag-max:** Maximum record lag. How far the consumer is behind producers. In a situation where real-time processing is important, high lag might mean you need more consumers.
- **bytes-consumed-rate:** Rate of bytes consumed per second. Gives a good idea of throughput.
- **records-consumed-rate:** Rate of records consumed per second.
- **fetch-rate:** Fetch requests per second. If this falls suddenly or goes to zero, it may be an indication of problems with the consumer.

Producer Tuning

Documentation

- **https://kafka.apache.org/documentation/#producerconfigs**

Important Concepts

You can tune your producers for better performance in your specific use case using a variety of configuration options.

Here are a few important configurations options for producers:

- **acks:** Determines when the broker will acknowledge the record.
  - 0: Producer will not wait for acknowledgement from the server. The record is considered acknowledged as soon as it is sent. The producer will not receive certain metadata such as the record offset.
  - 1: Record will be acknowledged when the leader writes the record to disk. Note that this creates a single point of failure. If the leader fails before followers replicate the record, data loss will occur.
  - all / -1: Record will be acknowledged only when the leader and all replicas have written the record to their disks. The acks may take longer, but this provides the maximum data integrity guarantee.
- **retries:** Number of times to retry a record if there is a transient error. If `max.in.flight.requests.per.connection` is not set to 1, the retry could cause records to appear in a different order. Transient errors: Temporary errors that is expected to resolved on their own. e.g. network glitches
- **batch.size:** Producers batch records sharing the same partition into a single request to create fewer requests. This specifies the maximum number of bytes in a batch. Messages larger than this size will not be batched. Requests can contain multiple batches (one for each partition) if data is going to more than one partition.

Consumer Tuning

Documentation

- **https://kafka.apache.org/documentation/#consumerconfigs**

Important Concepts

You can also tune consumers to optimize them for your use case.

Here are some important configuration options for consumers.

- **fetch.min.bytes:** The minimum amount of data to fetch in a request. If there is not enough data to satisfy this requirement, the request will wait for more data before responding. Set this higher to get better throughput in some situations at the cost of some latency.
- **heartbeat.interval.ms:** How often to send heartbeats to the consumer coordinator. Set this lower to allow a quicker rebalance response when a consumer joins or leaves the consumer group.
- **auto.offset.reset:** What to do when the consumer has no initial offset
  - latest: Start at the latest record.
  - earliest: Start with the earliest record.
  - none: Throw an exception when there is no existing offset data.
- **enable.auto.commit:** Periodically commit the current offset in the background. Use this to determine whether you want to handle offsets manually or automatically.

If more, then consumer will fetch more data in a single request.
Will not wait forever for the minimum amountof data to be available, there is also a time component to it.

Confluent KSQL

What Is KSQL?

Documentation

- **https://docs.confluent.io/current/ksql/docs/index.html**

Important Concepts

Confluent KSQL provides a SQL-like interface on top of Kafka Streams. With KSQL, you can do many of the same things you can do with a Kafka Streams application, such as:

- Data transformations
- Aggregations
- Joins
- Windowing
- Modeling data with streams and tables

KSQL looks similar to SQL, but it is not ANSI SQL compliant.

Also, while SQL select queries return data and stop running, KSQL select queries run continuously until stopped, streaming data in real time.

Using KSQL

Documentation

- **https://docs.confluent.io/current/ksql/docs/developer-guide/index.html**
- **https://docs.confluent.io/current/ksql/docs/tutorials/index.html**

Important Concepts

Here are some things you can do using KSQL.

List topics:

```
SHOW TOPICS;
```

Print records in a topic:

```
PRINT '<topic name>';
```

Create a stream from a topic:

```
CREATE STREAM <name> (<fields>) WITH (kafka_topic='<topic>', value_format='<format>');
```

Create a table from a topic:

```
CREATE TABLE <name> (<fields>) WITH (kafka_topic='<topic>', value_format='<format>', key='<key>');
```

Select from a table or stream:

```
SELECT <fields> FROM <table or stream>;
```

Aggregate data:

```
SELECT sum(<field>) FROM <table or stream> GROUP BY <field>;
```