Confluent Developer Skills for Building Apache Kafka®

Exercise Book

Version 5.3.0-v1.0.1

Table of Contents

Lab 01 Introduction

Copyright & Trademarks

Introduction

This document provides Hands-On Exercises for the course **Confluent Developer Training for Apache Kafka**. You will use a setup that includes a virtual machine (VM) configured as a Docker host to demonstrate the distributed nature of Apache Kafka.

The main Kafka cluster includes the following components, each running in a container:

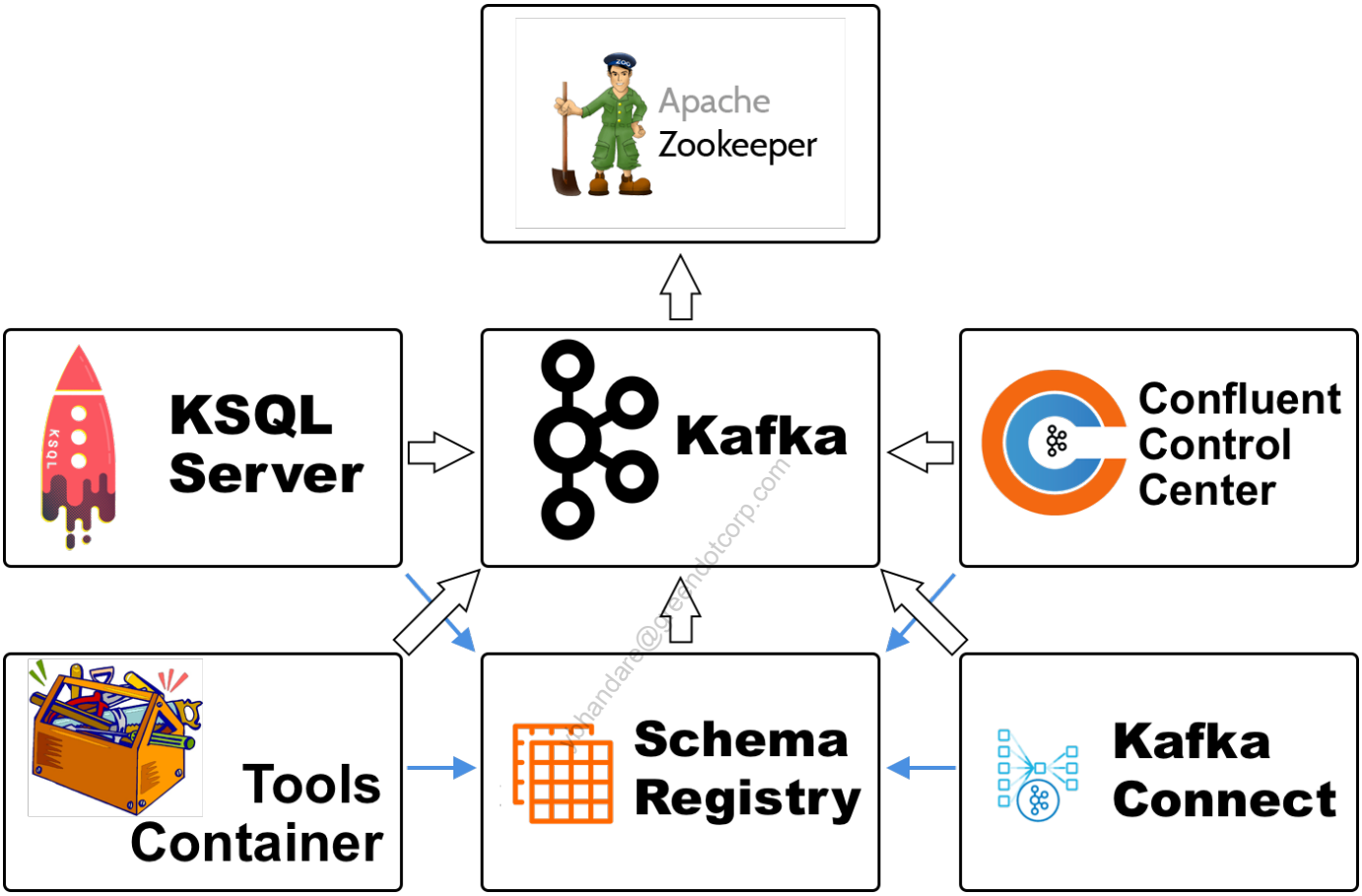


Table 1. Components of the Confluent Platform

Alias	Description
zookeeper	ZooKeeper
kafka	Kafka Broker
schema-registry	Schema Registry
connect	Kafka Connect
ksql-server	KSQL Server
control-center	Confluent Control Center
tools	secondary location for tools run against the cluster

You will use Confluent Control Center to monitor the main Kafka cluster. To achieve this, we are also running the Control Center service which is backed by the same Kafka cluster.

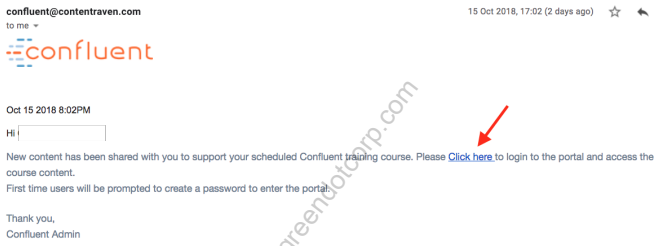
In this course we are using Confluent Platform version 5.3.0 which includes Kafka 2.3.0.

In production, Control Center should be deployed with its own dedicated Kafka cluster, separate from the cluster with production traffic. Using a dedicated metrics cluster is more resilient because it continues to provide system health monitoring even if the production traffic cluster experiences issues.

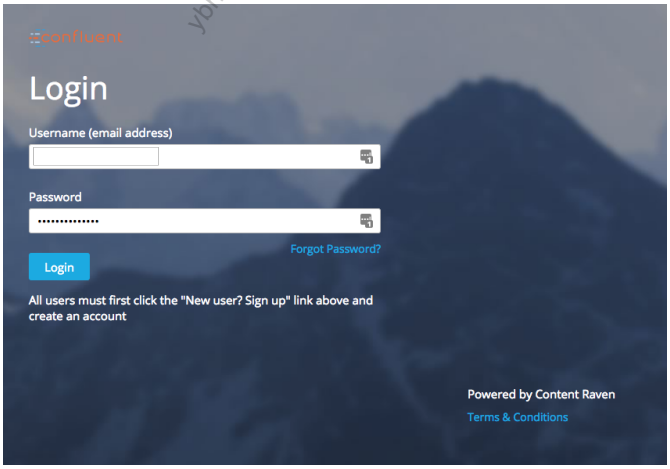
Accessing your Lab Environment

You will receive an email with a link to your lab environment from Confluent. The labs are running on a VM which is based on Ubuntu 18.04 Desktop edition. On it we have installed Docker Community Edition (CE), Google Chrome and Visual Studio Code.

1. Click the link in the email you received from Confluent:

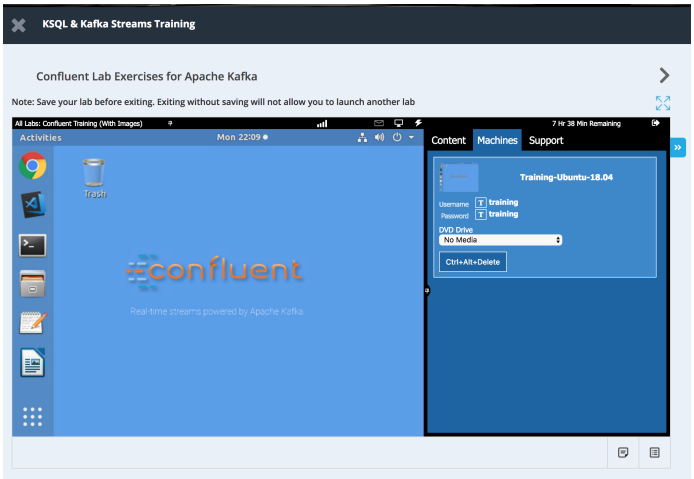


2. Login (or signup) to Content Raven:



3. Access your Learning Path
4. Launch the accompanying VM
5. Login to the VM using the following credentials:
    - **Username:** training

- **Password:** training



While you are working with the lab VM there might under certain circumstance a popup show up and ask you to install the latests updates (for Ubuntu). Please ignore this request!

Alternatively you can also download the VM to your laptop and run it in VirtualBox. Make sure you have the newest version of VirtualBox installed. Download the VM from this link: https://s3.amazonaws.com/confluent-training-images-us-east-1/training-ubuntu-18.04-jun2019.ova

If you have installed Docker for Desktop on your Mac or Windows 10 Pro machine then you can run the labs there. But please note that your trainer might not be able to troubleshoot any potential problems if you are running the labs locally. If you choose to do this, follow the instructions at → Running Labs in Docker for Desktop.

Preparing the Labs

1. Open a new terminal window
2. Download all Docker images needed in this course with the following script:

This step is only needed if you're not running the VM in the cloud but rather executing the labs in Docker for Desktop on your computer. This may take a couple of minutes!

```
$ CP_VERSION=5.3.0
for IMAGE in cp-zookeeper cp-enterprise-kafka cp-schema-registry \
             cp-kafka-connect cp-ksql-server cp-ksql-cli \
             cp-kafka-rest cp-enterprise-control-center \
             cp-kafkacat
do
    docker image pull confluentinc/${IMAGE}:${CP_VERSION}
done

5.3.0: Pulling from confluentinc/cp-zookeeper
Digest: sha256:f5cd4ac4782da1de36f263ecae0ec2d0894c96abe97fb76fd757cc1df9373a77
...
```

3. Clone the source code repository to the folder `confluent-dev` in your **home** directory:

```
$ cd ~
$ git clone --branch 5.3.0-v1.0.1 \
    https://github.com/confluentinc/training-developer-src.git \
    confluent-dev
```

If you chose to select another folder for the labs then note that many of our samples assume that the lab folder is `~/confluent-dev`. You will have to adjust all those command to fit your specific environment.

4. Navigate to the `labs` subfolder of the `confluent-dev` folder and start the Kafka cluster:

```
$ cd ~/confluent-dev/labs
$ docker-compose up -d
```

You should see something similar to this:

```
Creating network "labs_confluent" with the default driver
Creating labs_control-center_1  ... done
Creating labs_schema-registry_1 ... done
Creating labs_connect_1         ... done
Creating labs_ksql-server_1     ... done
Creating labs_zookeeper_1       ... done
Creating labs_tools_1           ... done
Creating labs_kafka_1           ... done
```


5. Monitor the cluster with:

```
$ watch docker-compose ps

     Name                   Command            State                 Ports
-------------------------------------------------------------------------------------
labs_connect_1           /etc/confluent/docker/run   Up      0.0.0.0:8083->8083/tcp,
9092/tcp
labs_control-center_1    /etc/confluent/docker/run   Up      0.0.0.0:9021->9021/tcp
labs_kafka_1             /etc/confluent/docker/run   Up      0.0.0.0:9092->9092/tcp
labs_ksql-server_1       /etc/confluent/docker/run   Up      0.0.0.0:8088->8088/tcp
labs_schema-registry_1   /etc/confluent/docker/run   Up      0.0.0.0:8081->8081/tcp
labs_tools_1             /bin/sh                     Up      8083/tcp, 9092/tcp
labs_zookeeper_1         /etc/confluent/docker/run   Up      2181/tcp, 2888/tcp, 3888/tcp
```

All services should have `State` equal to `Up`. Press `CTRL-c` to end watching.

If you are on a Mac you might have to first install the `watch` tool using e.g. **Homebrew**.

6. You can also observe the stats of Docker on your VM:

```
$ docker stats

CONTAINER ID        NAME                CPU %         MEM USAGE / LIMIT    MEM %    ...
9f911b165c44        labs_kafka_1        2.03%         380.3MiB / 7.79GiB   4.77%
f33b20e584b1        labs_tools_1        0.00%         544KiB / 7.79GiB     0.01%
3aaf4a2165c0        labs_ksql-server_1  1.01%         218.2MiB / 7.79GiB   2.74%
5c09a1a70b7c        labs_connect_1      3.70%         1.692GiB / 7.79GiB   21.72%
cd4c260b2c75        labs_zookeeper_1    0.40%         66.38MiB / 7.79GiB   0.83%
792bec1edd97        labs_schema-registry0.87%         168.1MiB / 7.79GiB   2.11%
fa537b8d2d83        labs_control-center_2.37%         361.5MiB / 7.79GiB   4.53%
```

Press CTRL-C to exit the Docker statistics.

Testing the Installation

1. Use the `zookeeper-shell` command to verify that all Brokers have registered with ZooKeeper. You should see a single Broker listed as `[101]` in the last line of the output.

```
$ zookeeper-shell zookeeper:2181 ls /brokers/ids
Connecting to zookeeper:2181

WATCHER::

WatchedEvent state:SyncConnected type:None path:null
[101]
```


OPTIONAL: Analyzing the Docker Compose File

1. Open the file `docker-compose.yml` in your editor and:
   a. locate the various services that are listed in the table earlier in this section
   b. note that the `alias` (e.g. `zookeeper` or `kafka`) are used to resolve a particular service
   c. note how the broker (kafka)
      i. gets a unique ID assigned via environment variable KAFKA_BROKER_ID
      ii. defines where to find the ZooKeeper instance

```
KAFKA_ZOOKEEPER_CONNECT: zookeeper:2181
```

      iii. sets the replication factor for the offsets topic to 1:

```
KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 1
```

iv. configures the broker to send metrics to Confluent Control Center:

```
KAFKA_METRIC_REPORTERS: "io.confluent.metrics.reporter.ConfluentMetricsReporter"
CONFLUENT_METRICS_REPORTER_BOOTSTRAP_SERVERS: "kafka:9092"
```

d. note how various services use the environment variable ..._BOOTSTRAP_SERVERS to define the list of Kafka brokers that serve as bootstrap servers (in our case it's only one instance):

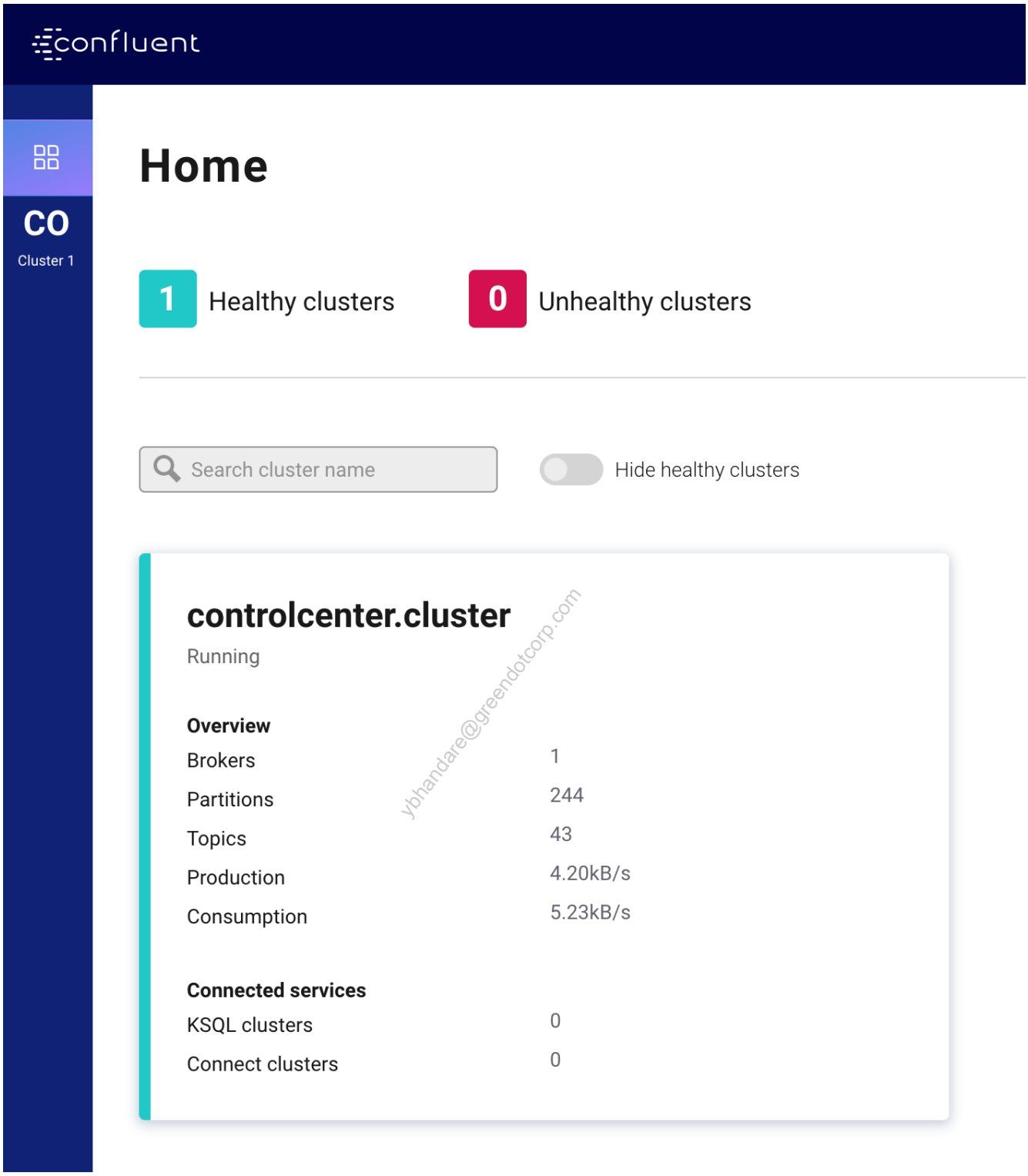
```
..._BOOTSTRAP_SERVERS: kafka:9092
```

e. note how e.g. the `connect` service and the `ksql-server` service define producer and consumer interceptors that produce data which can be monitored in Confluent Control Center:

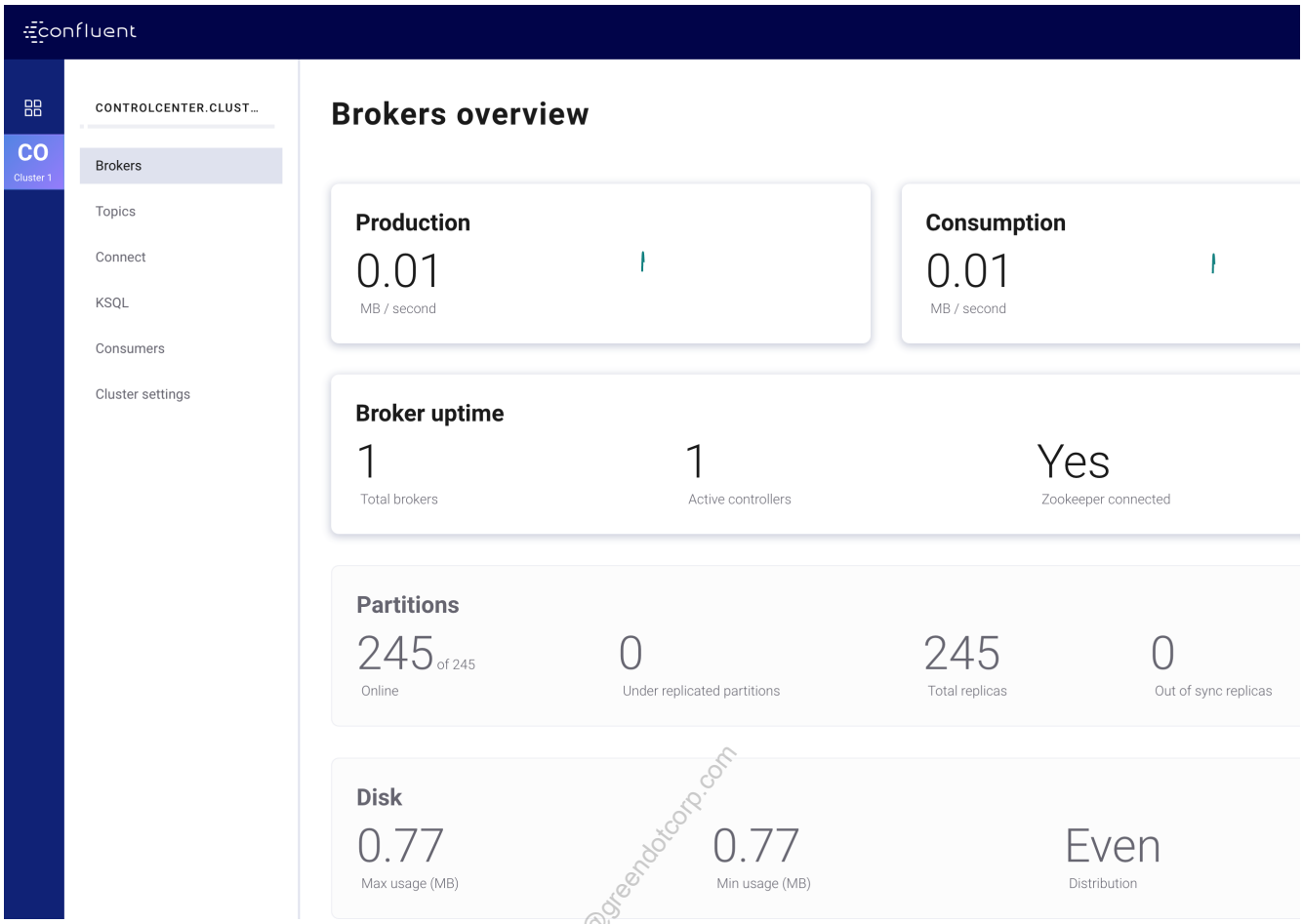
```
io.confluent.monitoring.clients.interceptor.MonitoringProducerInterceptor
io.confluent.monitoring.clients.interceptor.MonitoringConsumerInterceptor
```

Using Confluent Control Center

1. On your host machine, open a new browser tab in Google Chrome.
2. Navigate to Control Center at the URL `http://localhost:9021`:



3. Select the cluster **CO** and you will see this:



We have a single broker in our cluster. Also note the other important metrics of our Kafka cluster on this view.

4. Optional: Explore the other tabs of Confluent Control Center, such as **Topics** or **Cluster Settings**.

Running the Confluent Platform in Kubernetes

For more information on how to run Confluent OSS or Confluent Enterprise in Kubernetes please refer to the following links:

1. Confluent Platform Helm Charts: https://docs.confluent.io/current/quickstart/cp-helm-charts/docs/index.html
2. Helm Charts on GitHub: https://github.com/confluentinc/cp-helm-charts

Command Line Examples

Most Exercises contain commands that must be run from the command line. These commands will look like this:

```
$ pwd
/home/training
```

Commands you should type are shown in **bold**; non-bold text is an example of the output produced as a result of the command.

Lab 02 Using Kafka's Command-Line Tools

Using Kafka's Command-Line Tools

In this Hands-On Exercise you will start to become familiar with some of Kafka's command-line tools. Specifically you will:

- Use a tool to **create** a topic
- Use a console program to **produce** a message
- Use a console program to **consume** a message
- Use a tool to explore data stored in ZooKeeper

Prerequisites

1. Navigate to the labs folder:

   ```
   $ cd ~/confluent-dev/labs
   ```

2. Run the Kafka cluster, including Confluent Control Center:

   ```
   $ docker-compose up -d
   ```

3. Make sure all services are up and running:

   ```
   $ docker-compose ps
   ```

   You should see an output similar to this:

   ```
            Name                       Command            State                 Ports
   ------------------------------------------------------------------------------------------
   -----
   labs_connect_1           /etc/confluent/docker/run   Up       0.0.0.0:8083->8083/tcp,
   9092/tcp
   labs_control-center_1    /etc/confluent/docker/run   Up       0.0.0.0:9021->9021/tcp
   labs_kafka_1             /etc/confluent/docker/run   Up       0.0.0.0:9092->9092/tcp
   labs_ksql-cli_1          /bin/sh                     Up
   labs_ksql-server_1       /etc/confluent/docker/run   Up       0.0.0.0:8088->8088/tcp
   labs_schema-registry_1   /etc/confluent/docker/run   Up       0.0.0.0:8081->8081/tcp
   labs_tools_1             /bin/sh                     Up
   labs_zookeeper_1         /etc/confluent/docker/run   Up       2181/tcp, 2888/tcp,
   3888/tcp
   ```

Console Producing and Consuming

Kafka has built-in command line utilities to produce messages to a Topic and read messages from a Topic. These are extremely useful to verify that Kafka is working correctly, and for testing and debugging.

1. Before we can start writing data to a topic in Kafka, we need to first create that topic using a tool called `kafka-topics`. From within the terminal window run the command:

   ```
   $ kafka-topics
   ```

   This will bring up a list of parameters that the `kafka-console-producer` program can receive. Take a moment to look through the options.

2. Now execute the following command to create the topic `testing`:

   ```
   $ kafka-topics --bootstrap-server kafka:9092 \
       --create \
       --partitions 1 \
       --replication-factor 1 \
       --topic testing
   ```

   We create the topic with a single partition and `replication-factor` of one.

   We could have configured Kafka to allow **auto-creation** of topics. In this case we would not have had to do the above step and the topic would automatically be created when the first record is written to it. But this behavior is **strongly discouraged** in production. Always create your topics explicitly!

3. Now let's move on to start writing data into the topic just created. From within the terminal window run the command:

   ```
   $ kafka-console-producer
   ```

   This will bring up a list of parameters that the `kafka-console-producer` program can receive. Take a moment to look through the options. We will discuss many of their meanings later in the course.

4. Run `kafka-console-producer` again with the required arguments:

   ```
   $ kafka-console-producer --broker-list kafka:9092 --topic testing
   ```

   The tool prompts you with a >.

5. At this prompt type:

```
> some data
```

And hit **Enter**.

6. Now type:

```
> more data
```

And hit **Enter**.

7. Type:

```
> final data
```

And hit **Enter**.

8. Press `Ctrl-d` to exit the `kafka-console-producer` program.
9. Now we will use a Consumer to retrieve the data that was produced. Run the command:

```
$ kafka-console-consumer
```

This will bring up a list of parameters that the `kafka-console-consumer` can receive. Take a moment to look through the options.

10. Run `kafka-console-consumer` again with the following arguments:

```
$ kafka-console-consumer \
    --bootstrap-server kafka:9092 \
    --from-beginning \
    --topic testing
```

After a short moment you should see all the messages that you produced using `kafka-console-producer` earlier:

```
some data
more data
final data
```

11. Press `CTRL-c` to exit `kafka-console-consumer`.

OPTIONAL: Running producer and consumer in parallel

The `kafka-console-producer` and `kafka-console-consumer` programs can be run at the same time. Run

`kafka-console-producer` and `kafka-console-consumer` in separate terminal windows at the same time to see how `kafka-console-consumer` receives the events.

OPTIONAL: Working with record keys

By default, `kafka-console-producer` and `kafka-console-consumer` assume null keys. They can also be run with appropriate arguments to write and read keys as well as values.

1. Re-run the Producer with additional arguments to write (key,value) pairs to the Topic:

```
$ kafka-console-producer \
    --broker-list kafka:9092 \
    --topic testing \
    --property parse.key=true \
    --property key.separator=,
```

2. Enter a few values such as:

```
> 1,my first record
> 2,another record
> 3,Kafka is cool
```

3. Press `CTRL-d` to exit the producer.
4. Now run the **Consumer** with additional arguments to print the key as well as the value:

```
$ kafka-console-consumer \
    --bootstrap-server kafka:9092 \
    --from-beginning \
    --topic testing \
    --property print.key=true

null    some data
null    more data
null    final data
1   my first record
2   another record
3   Kafka is cool
```

Note the `NULL` values for the first 3 records that we entered earlier...

5. Press `CTRL-c` to exist the consumer.

The ZooKeeper Shell

1. Kafka's data in ZooKeeper can be accessed using the `zookeeper-shell` command:

```
$ zookeeper-shell zookeeper
Connecting to zookeeper
Welcome to ZooKeeper!
JLine support is disabled

WATCHER::

WatchedEvent state:SyncConnected type:None path:null
[zk: zookeeper(CONNECTED) 0]
```

2. From within the `zookeeper-shell` application, type `ls /` to view the directory structure in ZooKeeper. Note the `/` is required.

```
ls /
[schema_registry, cluster, controller, controller_epoch, brokers, zookeeper, admin,
isr_change_notification, consumers, log_dir_event_notification, latest_producer_id_block,
config]
ls /brokers
[ids, topics, seqid]
ls /brokers/ids
[101]
```

   Note the last output `[101]`, indicating that we have a single broker with ID `101` in our cluster.

3. Press `CTRL-d` to exit the ZooKeeper shell.

Cleanup

1. Execute the following command to completely cleanup your environment:

```
$ docker-compose down -v
```

Conclusion

In this lab you have used Kafka command line tools to create a topic, write and read from this topic. Finally you have used the ZooKeeper shell tool to access data stored within ZooKeeper.

STOP HERE. THIS IS THE END OF THE EXERCISE.

Lab 03 Consuming from Multiple Partitions

Consuming from Multiple Partitions

In this Hands-On Exercise, you will create a topic with **multiple partitions**, produce data to those partitions, and then read it back to observe issues with ordering.

Prerequisites

1. Navigate to the labs folder:

```
$ cd ~/confluent-dev/labs
```

2. Run the Kafka cluster, including Confluent Control Center:

```
$ docker-compose up -d
```

Consuming from multiple Partitions

1. From within the terminal window create a Topic manually with Kafka's command-line tool, specifying that it should have **two partitions**:

```
$ kafka-topics \
    --bootstrap-server kafka:9092 \
    --create \
    --topic two-p-topic \
    --partitions 2 \
    --replication-factor 1
```

2. You can use the `kafka-topics` tool to describe the details of the topic:

```
$ kafka-topics \
    --bootstrap-server kafka:9092 \
    --describe \
    --topic two-p-topic
```

   Which will result in this output:

```
Topic:two-p-topic   PartitionCount:2    ReplicationFactor:1 Configs:
    Topic: two-p-topic  Partition: 0    Leader: 101 Replicas: 101   Isr: 101
    Topic: two-p-topic  Partition: 1    Leader: 101 Replicas: 101   Isr: 101
```

3. Use the command-line Producer to write several lines of data to the Topic.

```
$ seq 1 100 > numlist &&
  kafka-console-producer \
    --broker-list kafka:9092 \
    --topic two-p-topic < numlist
```

4. Use the command-line Consumer to read the Topic

```
$ kafka-console-consumer \
    --bootstrap-server kafka:9092 \
    --from-beginning \
    --topic two-p-topic
```

   Press CTRL-C to exit the consumer.

5. Note the order of the numbers. Rerun the Producer command in step 3, then rerun the Consumer command in step 4 and see if you observe any difference in the output.
6. What do you think is happening as the data is being written?
7. Try creating a new Topic with three Partitions and running the same steps again.

Cleanup

1. Execute the following command to completely cleanup your environment:

```
$ docker-compose down -v
```


Conclusion

You have been creating a topic with multiple partitions. Then you used the `kafka-console-producer` to write some data to the topic. Finally you analyzed the order of the data output by the `kafka-console-consumer` and noticed that the order is not deterministic.

STOP HERE. THIS IS THE END OF THE EXERCISE.

Lab 04a Creating a Kafka Producer (Java)

Creating a Kafka Producer in Java

The goal of this lab is to create a simple producer, that ingest live data of the **High-frequency Positioning** service of **digitransit** into Kafka. The data is published via MQTT interface. You will write the producer in Java.

For more details about the MQTT service see: https://digitransit.fi/en/developers/apis/4-realtime-api/vehicle-positions/

Prerequisites

1. Navigate to the project folder:

```
$ cd ~/confluent-dev/labs/create-producer-java
```

2. If you haven't done so already, run the Kafka cluster, including Confluent Control Center:

```
$ docker-compose up -d
```

3. Create the topic `vehicle-positions` with 6 partitions on Kafka with this command:

```
$ kafka-topics \
    --create \
    --bootstrap-server kafka:9092 \
    --partitions 6 \
    --replication-factor 1 \
    --topic vehicle-positions
```


4. Open the project in Visual Studio Code:

```
$ code .
```

Writing the Producer

1. Notice that there are the two classes `VehiclePositionsProducer.java` and `Subscriber.java` in the folder `src/main/java/clients` of the project folder. Open them in the code editor. We have added the scaffolding code to it.
2. Let's start with the `VehiclePositionsProducer.java` implementation. First we have to define the configuration for the Kafka producer we want to create. Add the following code to the `main` method:

```
Properties settings = new Properties();
settings.put(ProducerConfig.CLIENT_ID_CONFIG, "vp-producer");
settings.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "kafka:9092");
settings.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, StringSerializer.class);
settings.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, StringSerializer.class);
```

Configurations for Kafka clients (producer, consumer, Kafka Streams applications, etc.) are always defined as objects of type `Properties`. Here are the details:

- We give the producer a `client.id`
- We define the property `bootstrap.servers`, that is where the producer can find the Kafka cluster. This is a list of one to several Kafka servers. In our simple case we only have a single broker at hand, so we pass only this one.
- The next 2 items define what serializers the producer shall use when serializing the `key` and `value` of a record before sending it to the broker. In both cases we use the `StringSerializer`.

3. Next we add a line of code to instantiate a producer object using the `settings` object defined above:

```
final KafkaProducer<String, String> producer = new KafkaProducer<>(settings);
```

The `KafkaProducer` class is generic in `key` and `value`. Since we're using `String` in both cases, that's how our producer is defined.

4. Now we define what shall happen if the application (the producer) shuts down expectedly or unexpectedly. We add the shutdown behavior:

```
Runtime.getRuntime().addShutdownHook(new Thread(() -> {
    System.out.println("### Stopping VP Producer ###");
    producer.close();
}));
```

As you can see, the main thing here is that we are closing the producer.

5. So far so good, but now we are implementing the code that receives IoT data from the **High-frequency Positioning** service of **digitransit**. This service is using the MQTT protocol and thus we have to use a Java MQTT client. In our sample we're using the popular Paho client (https://www.eclipse.org/paho/).
6. We are putting all the implementation details into the class `Subscriber.java`, thus add the following 2 lines at the end of the `main` method of the `VehiclePositionsProducer.java` class:

```
Subscriber subscriber = new Subscriber(producer);
subscriber.start();
```

We are instantiating a `subscriber` object and passing it the producer instance. Then we're starting the subscriber. With this the `VehiclePositionsProducer.java` class is complete and we can move on to the

`Subscriber.java` implementation.

7. Open the `Subscriber.java` class and notice the scaffolding there. This class implements the `MqttCallback` interface defined by the Paho MQTT library. First let's define the constructor:

```
private KafkaProducer<String, String> producer;

public Subscriber(KafkaProducer<String, String> producer) {
    this.producer = producer;
}
```

Notice how we're defining an instance variable for the producer and assign the constructor parameter to it.

8. Before we continue let's define a few additional instance variables as follows:

```
private final int qos = 1;
private String host = "ssl://mqtt.hsl.fi:8883";
private String clientId = "MQTT-Java-Example";
private String topic = "/hfp/v2/journey/ongoing/vp/#";
private String kafka_topic = "vehicle-positions";
private MqttClient client;
```


We have the following:

- `qos` defines the **quality of service** of the MQTT service
- `host` is the URI of the host providing the data, which we're going to connect to
- `clientId` is just how we identify ourselves to the host
- `topic` is the MQTT topic (**NOT** the Kafka topic!) we're going to subscribe to
- `kafka_topic` is the name of the Kafka topic we're going to write the MQTT data to
- Finally we are defining a variable `client` for the `MqttClient` instance

If you have problems with the amount of data that the MQTT source produces, you can adjust the value of `topic` to say only return vehicle positions for **trams**. The corresponding value would then be: `/hfp/v2/journey/ongoing/vp/tram/#`
For more details please consult https://digitransit.fi/en/developers/apis/4-realtime-api/vehicle-positions/

9. Let's now define the `start` method:

```
public void start() throws MqttException {
    MqttConnectOptions conOpt = new MqttConnectOptions();
    conOpt.setCleanSession(true);

    final String uuid = UUID.randomUUID().toString().replace("-", "");

    String clientId = this.clientId + "-" + uuid;
    this.client = new MqttClient(this.host, clientId, new MemoryPersistence());
    this.client.setCallback(this);
    this.client.connect(conOpt);

    this.client.subscribe(this.topic, this.qos);
}
```

We are instantiating a `MqttClient` instance, passing it the `host` and `clientId`, as well as the persistence method (here we use `MemoryPersistence`). Then we define which class (or object) to use for the callbacks triggered by the `MqttClient`. In our case we're using this very instance to handle the callbacks. Next we connect the client using the connection options define on the first two lines of the method. Finally we subscribe the client to the given topic and ask for the given quality of service.

10. Next we need to implement the 3 callback methods defined on the interface `MqttCallback`. Let's start with the most important one for us, the callback upon message arrival. Add this code snippet to the class:

```
public void messageArrived(String topic, MqttMessage message) throws MqttException {
    System.out.println(String.format("[%s] %s", topic, new String(message.
getPayload())));
    final String key = topic;
    final String value = new String(message.getPayload());
    final ProducerRecord<String, String> record =
        new ProducerRecord<>(this.kafka_topic, key, value);
    producer.send(record);
}
```


11. We also have to implement the callback triggered upon completion of the delivery. This method we just leave empty for this simple sample:

```
public void deliveryComplete(IMqttDeliveryToken token) {
}
```

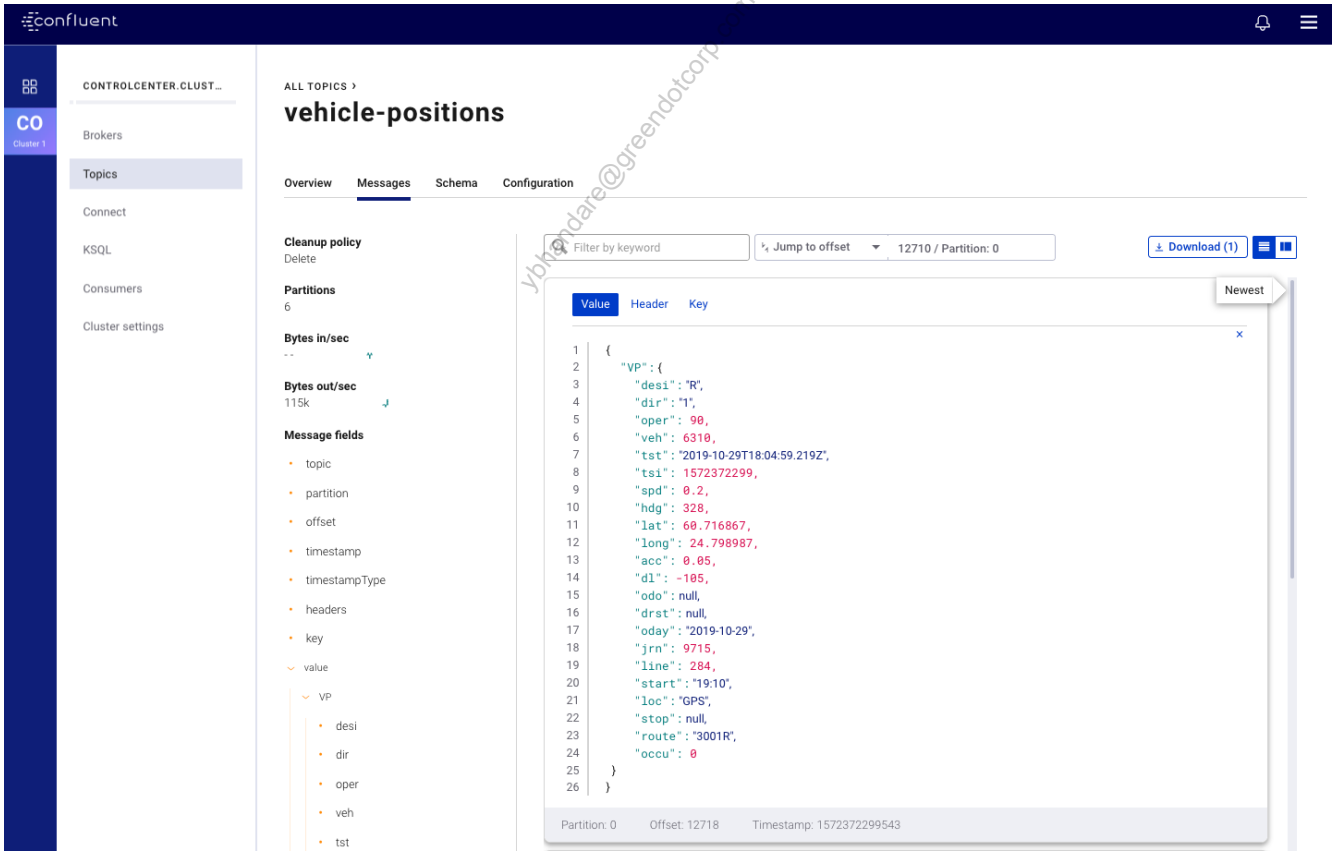
12. The third and last callback is triggered upon loss of the connection to the MQTT host. Add this code snippet to the class:

```
public void connectionLost(Throwable cause) {
    System.out.println("Connection lost because: " + cause);
    System.exit(1);
}
```

13. Now, in VS Code select the menu **Debug** → **Start Debugging**. Observe the following output in the **DEBUG CONSOLE**:

```
* Starting VP Producer *
...
[/hfp/v2/journey/ongoing/vp/tram/0040/00461/1003/1/Meilahti/19:34/1111428/4/60;24/19/75/7
0] {"VP":{"desi":"3","dir":"1","oper":40,"veh":461,"tst":"2019-10-
29T17:50:46.228Z","tsi":1572371446,"spd":2.86,"hdg":348,"lat":60.177718,"long":24.950000,
"acc":0.34,"dl":145,"odo":3512,"drst":0,"oday":"2019-10-
29","jrn":1577,"line":31,"start":"19:34","loc":"GPS","stop":null,"route":"1003","occu":0}
}
[/hfp/v2/journey/ongoing/vp/bus/0022/01171/4619/1/Simonsilta/19:35/4650221/4/60;25/30/12/
41] {"VP":{"desi":"619","dir":"1","oper":22,"veh":1171,"tst":"2019-10-
29T17:50:46.252Z","tsi":1572371446,"spd":8.26,"hdg":54,"lat":60.314735,"long":25.021490,"
acc":0.90,"dl":-99,"odo":5048,"drst":0,"oday":"2019-10-
29","jrn":312,"line":808,"start":"19:35","loc":"GPS","stop":null,"route":"4619","occu":0}
}
...
```

14. In your browser navigate to http://localhost:9021 to access Confluent Control Center. Select cluster **Cluster 1**. Then navigate to **Topics** and select the `vehicle-positions` topic. In the details view select the **Messages** tab. On the tab select **Card view** and expand one of the messages. You should see something like this:



You can view the details of an individual message, which contains the expected vehicle position record.

15. When done, stop the debugger in VS Code.

OPTIONAL: Building a Docker Image for the producer

To prepare for the next lab we want to create a Docker image containing our producer, so we can easily run it as a part of our distributed application.

1. In VS Code open the file `Dockerfile` in the project folder and observe the steps needed to containerize our Java application. Specifically note that, in order to reduce the size of the final image, we are using a multi-step build process.
2. In the terminal, from within the project folder execute the following command to build the image:

   ```
   $ docker image build -t cnfltraining/vp-producer:v2 .
   ```

   don't forget the period (`.`) at the end of the above command! It instructs the Docker builder to use the current directory as the context.

3. Test the new image by running a container from it:

   ```
   $ docker container run --rm -it \
       --net labs_confluent \
       cnfltraining/vp-producer:v2
   ```

   You should see an output like this (shortened):

   ```
   *** Starting VP Producer ***
   ...
   [/hfp/v2/journey/ongoing/vp/bus/0018/00279/4555/2/Keilaniemi
   (M)/19:21/2213220/5/60;24/18/72/59]
   {"VP":{"desi":"555","dir":"2","oper":18,"veh":279,"tst":"2019-10-
   29T18:10:07.596Z","tsi":1572372607,"spd":0.00,"hdg":193,"lat":60.175115,"long":24.829068,
   "acc":0.00,"dl":88,"odo":16005,"drst":1,"oday":"2019-10-
   29","jrn":474,"line":933,"start":"19:21","loc":"GPS","stop":2213220,"route":"4555","occu"
   :0}}
   [/hfp/v2/journey/ongoing/vp/train/0090/01071/3001I/1/Lentoasema-
   Helsinki/19:51/4610503/4/60;25/20/83/19]
   {"VP":{"desi":"I","dir":"1","oper":90,"veh":1071,"tst":"2019-10-
   29T18:10:07.666Z","tsi":1572372607,"spd":20.56,"hdg":13,"lat":60.281891,"long":25.039366,
   "acc":-2.11,"dl":-116,"odo":14123,"drst":0,"oday":"2019-10-
   29","jrn":9112,"line":279,"start":"19:51","loc":"GPS","stop":null,"route":"3001I","occu":
   0}}
   ...
   ```

4. Stop the producer by pressing `CTRL-C`.

The complete solution can be found at `~/confluent-dev/solution/create-producer-java`.

Cleanup

1. Execute the following command to completely cleanup your environment:

```
$ docker-compose down -v
```

Question

What would happen if the producer you have created here was being run on a server that failed? Can you think of a way to distribute this application and hand off processing if a failure were to occur?

We will revisit this topic in a future module on Kafka Connect.

Conclusion

In this exercise you have created a Kafka producer in Java that writes data, received from the **High-frequency Positioning** service of **digitransit**, into a Kafka topic called `vehicle-positions`.

STOP HERE. THIS IS THE END OF THE EXERCISE.

Lab 04b Creating a Kafka Producer (Node JS)

Creating a Kafka Producer in Node JS

The goal of this lab is to create a simple producer that ingest live data of the **High-frequency Positioning** service of **digitransit** into Kafka. The data is published via MQTT interface. You will write the producer in Node JS.

For more details about the MQTT service see: https://digitransit.fi/en/developers/apis/4-realtime-api/vehicle-positions/

This is an optional lab, for all those learners that like to work with another language than just Java. The result is the same as in the previous lab where we implement the producer in Java.

Prerequisites

1. Install Node on the VM:

```
$ curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0.34.0/install.sh | bash &&
source ~/.bashrc &&
nvm install lts/carbon
```

2. Install Python on the VM:

```
$ sudo apt update && sudo apt install -y python
```

3. Navigate to the project folder:

```
$ cd ~/confluent-dev/labs/create-producer-node
```

4. Run the Kafka cluster, including Confluent Control Center:

```
$ docker-compose up -d
```

5. Create the topic `vehicle-positions` with 6 partitions on Kafka with this command:

```
$ kafka-topics \
    --create \
    --bootstrap-server kafka:9092 \
    --partitions 6 \
    --replication-factor 1 \
    --topic vehicle-positions
```

If the above command triggers an error, then it could be either that the broker is **not yet ready** and needs a few additional moments to finish its startup, or it could be that the topic `vehicle-positions` already existed on the broker.

6. Open the project in Visual Studio Code:

```
$ code .
```

Writing the Producer

1. Open the file `server.js` and analyze its content. You find sample code on how to retrieve vehicle positions via MQTT from the **High-frequency Positioning** service. In this sample exactly 100 positions are retrieved until the program stops.
2. Initialize the application by running:

```
$ npm install
```

3. Run the application by pressing selecting the menu **Debug** → **Start Debugging** in VS Code, and observe the output in the integrated **DEBUG CONSOLE** window. You should see something like this (shortened):

```
/usr/local/bin/node --inspect-brk=19068 solution/create-producer/server.js
Debugger listening on ws://127.0.0.1:19068/6c895664-0027-4734-8a09-1b944e049d90
Debugger attached.
Connected
Route 84 (vehicle 22/1099): 60.195381,25.032492 - 0.09m/s
Route 322 (vehicle 22/1069): 60.256728,24.80961 - 1.35m/s
Route 909K (vehicle 6/158): 60.201413,24.338191 - 9.47m/s
Route 909K (vehicle 6/158): 60.201094,24.338395 - 5.25m/s
...
```


As you can see some real-time vehicle positions are output. In my example a vehicle on route `84` from the provider `22` with the vehicle number `1099` is at the geo position `60.195381,25.032492`, driving at a speed of `0.09m/s`.

4. Now we're going to add code that writes the vehicle positions into the topic `vehicle-positions` on Kafka.
5. If you're working on a MAC (natively and not on our lab VM; Mac OS High Sierra / Mojave) then run the following script:

```
$ export CPPFLAGS=-I/usr/local/opt/openssl/include && \
export LDFLAGS=-L/usr/local/opt/openssl/lib
```

Otherwise you will encounter errors during the download and rebuild of the Kafka client library (see: https://github.com/Blizzard/node-rdkafka).

6. Install the Kafka client for node:

```
$ npm install node-rdkafka
```

7. Open the file `package.json` and observe that the following line has been added to the `dependencies` node:

```
"node-rdkafka": "^2.7.1"
```

The version number might be different in your case.

8. To use the module, you must require it. Add the following line to `server.js`:

```
const Kafka = require('node-rdkafka');
```

9. Create a producer with the following code snippet:

```
const producer = new Kafka.Producer({
    'client.id': 'vp-producer',
    'metadata.broker.list': 'kafka:9092',
    'dr_cb': true
});
```

Indicating that our producer is called `vp-producer`, connects to the bootstrap server `kafka:9092`, and returns a delivery report (`'dr_cb': true`) on each send.

10. Set the producer's poll interval to 100 ms:

```
producer.setPollInterval(100);
```

11. Now connect the producer:

```
producer.connect();
```

12. Define the callback function for the delivery report:

```
producer.on('delivery-report', (err, report) => {
    // Report of delivery statistics here:
    console.log(report);
});
```

13. Let's add some error handling code:

```
// Any errors we encounter, including connection errors
producer.on('event.error', err => {
  console.error('Error from producer');
  console.error(err);
})
```

14. Use the following snippet to write the messages received via **MQTT** to Kafka (replacing the whole `client.on('message', …)` loop in `server.js`):

```
producer.on('ready', () => {
    client.on('message', (topic, message) => {
        try {
            const vehicle_position = JSON.parse(message);
            const key = topic;
            const value = JSON.stringify(vehicle_position);
            producer.produce(
                'vehicle-positions',
                null,
                Buffer.from(value),
                key,
                Date.now()
            );
        } catch (err) {
            client.end(true);
            console.error('A problem occurred when sending our message');
            console.error(err);
        }
    });
});
```


Note how we define the value of the Kafka record as the message we received from MQTT. The key of the record is defined as the combination of vehicle operator and vehicle number. We also add the current timestamp to the record.

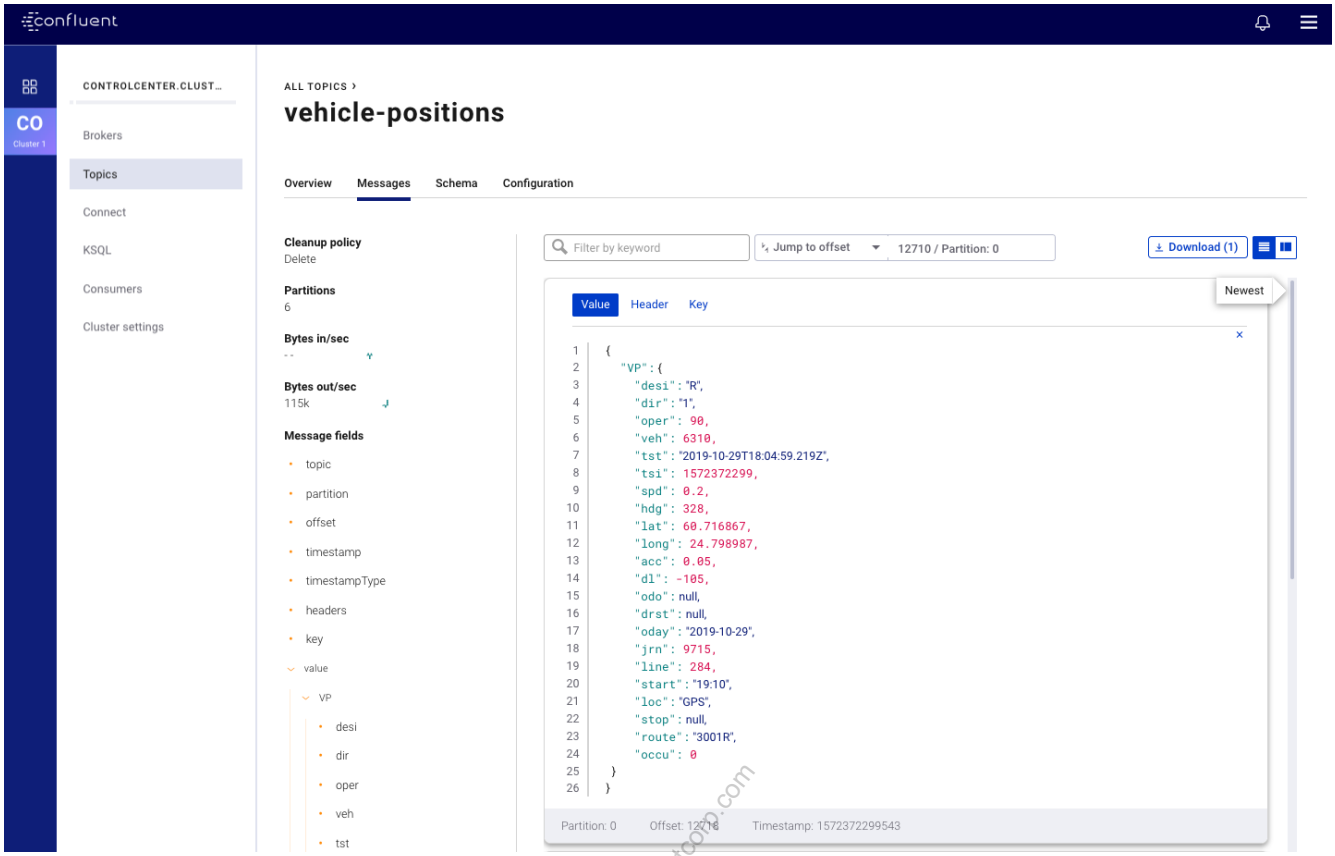
15. Run the application with **Debug → Start Debugging**.
16. Use the `kafka-console-consumer` to observe the inflowing data in Kafka:

```
$ kafka-console-consumer \
    --bootstrap-server kafka:9092 \
    --topic vehicle-positions \
    --from-beginning \
    --max-messages 10
```

which should result in something similar to this:

```
{"VP":{"desi":"40","dir":"1","oper":12,"veh":1906,"tst":"2019-10-
29T21:11:49.137Z","tsi":1572383509,"spd":4.32,"hdg":19,"lat":60.220097,"long":24.901856,"
acc":-1.87,"dl":-168,"odo":null,"drst":null,"oday":"2019-10-
29","jrn":980,"line":62,"start":"22:52","loc":"GPS","stop":null,"route":"1040","occu":0}}
{"VP":{"desi":"562","dir":"2","oper":22,"veh":819,"tst":"2019-10-
29T21:11:49.164Z","tsi":1572383509,"spd":11.16,"hdg":199,"lat":60.258746,"long":25.083674
,"acc":0.09,"dl":-102,"odo":554,"drst":null,"oday":"2019-10-
29","jrn":733,"line":657,"start":"22:30","loc":"GPS","stop":null,"route":"4562","occu":0}
}
{"VP":{"desi":"52","dir":"2","oper":18,"veh":817,"tst":"2019-10-
29T21:11:49.131Z","tsi":1572383509,"spd":9.46,"hdg":41,"lat":60.219139,"long":24.900615,"
acc":-1.41,"dl":-275,"odo":6159,"drst":0,"oday":"2019-10-
29","jrn":1456,"line":68,"start":"22:52","loc":"GPS","stop":null,"route":"1052","occu":0}
}
{"VP":{"desi":"74N","dir":"2","oper":30,"veh":33,"tst":"2019-10-
29T21:11:48.708Z","tsi":1572383508,"spd":9.16,"hdg":167,"lat":60.196806,"long":24.962178,
"acc":0.11,"dl":58,"odo":14213,"drst":0,"oday":"2019-10-
29","jrn":276,"line":101,"start":"22:43","loc":"GPS","stop":null,"route":"1074N","occu":0
}}
{"VP":{"desi":"739","dir":"2","oper":22,"veh":1178,"tst":"2019-10-
29T21:11:49.174Z","tsi":1572383509,"spd":8.75,"hdg":202,"lat":60.339275,"long":25.07994,"
acc":-0.42,"dl":-156,"odo":8335,"drst":0,"oday":"2019-10-
29","jrn":525,"line":774,"start":"22:53","loc":"GPS","stop":null,"route":"4739","occu":0}
}
{"VP":{"desi":"43","dir":"1","oper":12,"veh":1004,"tst":"2019-10-
29T21:11:49.168Z","tsi":1572383509,"spd":0,"hdg":138,"lat":60.171699,"long":24.939522,"ac
c":0,"dl":null,"odo":0,"drst":0,"oday":"2019-10-
29","jrn":911,"line":65,"start":"23:22","loc":"GPS","stop":null,"route":"1043","occu":0}}
{"VP":{"desi":"71","dir":"2","oper":55,"veh":1219,"tst":"2019-10-
29T21:11:49.157Z","tsi":1572383509,"spd":3.62,"hdg":175,"lat":60.234584,"long":25.009811,
"acc":0.13,"dl":-60,"odo":2975,"drst":0,"oday":"2019-10-
29","jrn":1856,"line":94,"start":"23:05","loc":"GPS","stop":null,"route":"1071","occu":0}
}
{"VP":{"desi":"665K","dir":"1","oper":18,"veh":2996,"tst":"2019-10-
29T21:11:49.188Z","tsi":1572383509,"spd":0,"hdg":15,"lat":60.473872,"long":25.089801,"acc
":0,"dl":240,"odo":9340,"drst":0,"oday":"2019-10-
29","jrn":35,"line":1040,"start":"22:55","loc":"GPS","stop":9500236,"route":"9665K","occu
":0}}
{"VP":{"desi":"415","dir":"2","oper":12,"veh":1816,"tst":"2019-10-
29T21:11:49.151Z","tsi":1572383509,"spd":19.06,"hdg":200,"lat":60.300962,"long":24.960133
,"acc":2.25,"dl":-30,"odo":2755,"drst":0,"oday":"2019-10-
29","jrn":600,"line":339,"start":"23:05","loc":"GPS","stop":null,"route":"4415","occu":0}
}
{"VP":{"desi":"104","dir":"1","oper":18,"veh":626,"tst":"2019-10-
29T21:11:49.111Z","tsi":1572383509,"spd":11.23,"hdg":348,"lat":60.164592,"long":24.773387
,"acc":-0.05,"dl":94,"odo":7821,"drst":0,"oday":"2019-10-
29","jrn":63,"line":1051,"start":"23:01","loc":"GPS","stop":null,"route":"2104","occu":0}
}
Processed a total of 10 messages
```


17. In your browser navigate to http://localhost:9021 to access Confluent Control Center. Select cluster **Cluster 1**. Then navigate to **Topics** and select the `vehicle-positions` topic. In the details view select the **Messages** tab. On the tab select **Card view** and expand one of the messages. You should see something like this:



You can view the details of an individual message, which contains the expected vehicle position record.

18. Stop the producer in VS Code.

Building a Docker Image for the producer

To prepare for the next lab we want to create a Docker image containing our producer, so we can easily run it as a part of our distributed application.

1. In VS Code open the file `Dockerfile` in the project folder and observe the steps needed to containerize our Node JS application.
2. In the terminal, from within the project folder execute the following command to build the image:

```
$ docker image build -t cnfltraining/vp-producer-node:v2 .
```

don't forget the period (`.`) at the end of the above command! It instructs the Docker builder to use the current directory as the context.

3. Test the new image by running a container from it:

```
$ docker container run --rm -it \
    --net labs_confluent \
    cnfltraining/vp-producer-node:v2
```

You should see an output like this (shortened):

```
> create-producer@1.0.1 start /app
> node server.js
...
Connected
{ topic: 'vehicle-positions',
  partition: 1,
  offset: 13745,
  key:
   <Buffer 2f 68 66 70 2f 76 32 2f 6a 6f 75 72 6e 65 79 2f 6f 6e 67 6f 69 6e 67 2f 76 70
2f 62 75 73 2f 30 30 31 38 2f 30 30 37 31 37 2f 36 31 37 33 4b 2f 31 2f ... >,
  timestamp: 1572385131174,
  size: 300 }
{ topic: 'vehicle-positions',
  partition: 2,
  offset: 13607,
  key:
   <Buffer 2f 68 66 70 2f 76 32 2f 6a 6f 75 72 6e 65 79 2f 6f 6e 67 6f 69 6e 67 2f 76 70
2f 62 75 73 2f 30 30 31 32 2f 30 31 35 30 38 2f 34 35 36 30 2f 31 2f 4d ... >,
  timestamp: 1572385131178,
  size: 302 }
...
```


4. Stop the producer by pressing CTRL-C.

The complete solution can be found at `~/confluent-dev/solution/create-producer-node`.

Cleanup

1. Execute the following command to completely cleanup your environment:

```
$ docker-compose down -v
```

Conclusion

In this exercise you have created a Kafka producer in Node JS that writes data, received from the **High-frequency Positioning** service of **digitransit**, into a Kafka topic called `vehicle-positions`.

STOP HERE. THIS IS THE END OF THE EXERCISE.

Lab 04c Creating a Kafka Consumer

Creating a Kafka Consumer

The goal of this lab is to create a simple Kafka consumer in Java, that consumes records from a topic called `vehicle-positions`. Those records are produced by a Kafka producer that ingests live data of the **High-frequency Positioning** service of **digitransit** into said Kafka topic.

- First you will create a consumer in Java
- Optional: create the same consumer in Python
- Optional: create the same consumer in .NET

Prerequisites

1. Navigate to the project folder:

```
$ cd ~/confluent-dev/labs/create-consumer-java
```

2. Run the Kafka cluster, including Confluent Control Center:

```
$ docker-compose up -d
```


3. Create the topic `vehicle-positions` with 6 partitions on Kafka with this command:

```
$ kafka-topics \
    --create \
    --bootstrap-server kafka:9092 \
    --partitions 6 \
    --replication-factor 1 \
    --topic vehicle-positions
```

4. Run a Docker container with the Kafka producer from the previous lab:

```
$ docker container run --rm -d \
    --name vp-producer \
    --hostname vp-producer \
    --net labs_confluent \
    cnfltraining/vp-producer:v2
```

If you were not able to finish the lab **Creating a Kafka Producer** then don't worry. The necessary image `cnfltraining/vp-producer:v2` will just be downloaded from our Confluent Training org on Docker Hub.

Alternatively you could also do the following in a new terminal window:

```
$ cd ~/confluent-dev/labs/create-producer-java && \
  gradle run
```

Or if you did not succeed with the previous lab then use the version from the solution folder:

```
$ cd ~/confluent-dev/solution/create-producer-java && \
  gradle run
```

If you encounter problems with the producer, e.g. the MQTT datasource that the producer is relying on is down, then please use the Docker image `cnfltraining/vp-producer-fallback:v2` instead.

5. Open the project in Visual Studio Code:

```
$ code .
```


Writing the Consumer

1. In VS Code open the file `build.gradle`. This is the project file for our Java based Kafka client. Specifically have a look at the node `dependencies`, it references the 3 external libraries the app will be using.
2. Now open the file `src/main/java/clients/VehiclePositionConsumer.java`. This file contains the scaffolding for our consumer, namely all the necessary import statements.
3. To this file add code to define a properties object containing the consumer settings:

```
Properties settings = new Properties();
settings.put(ConsumerConfig.GROUP_ID_CONFIG, "vp-consumer");
settings.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "kafka:9092");
settings.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest");
settings.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, StringDeserializer.class);
settings.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG, StringDeserializer.class);
```

4. Next we create a consumer instance:

```
KafkaConsumer<String, String> consumer = new KafkaConsumer<>(settings);
```

5. Then we define the actual poll loop, including the record handling:

```
try {
    consumer.subscribe(Arrays.asList("vehicle-positions"));
    while (true) {
        ConsumerRecords<String, String> records = consumer.poll(Duration.ofMillis(100));
        for (ConsumerRecord<String, String> record : records)
            System.out.printf("offset = %d, key = %s, value = %s\n",
                              record.offset(), record.key(), record.value());
    }
}
finally{
    System.out.println("*** Ending VP Consumer ***");
    consumer.close();
}
```

In the above code snippet we subscribe to the topic `vehicle-positions`. Then we have an endless `while` loop, inside which we `poll` Kafka using a timeout time of `100 ms`. Kafka returns us (a potentially empty) list of records. We then loop over those records, and for each of the records we output its `offset`, `key`, and `value` to the console.
Notice that the whole thing is in a `try` block such as that in the `finally` block we have the opportunity to clean up behind ourselves. There we mainly close the consumer.

6. Run the application by selecting the menu **Debug** → **Start Debugging** in VS Code.
   The output should look similar to this (shortened):


```
> Task :run
* Starting VP Consumer *
offset = 1117, key =
/hfp/v2/journey/ongoing/vp/metro/0050/00175/31M1/2/Matinkylä/19:55/1431604/3/60;25/20/04/
54, value = {"VP":{"desi":"M1","dir":"2","oper":50,"veh":175,"tst":"2019-10-
29T18:03:28.724Z","tsi":1572372208,"spd":9.18,"hdg":232,"lat":60.20571237,"long":25.04446
321,"acc":null,"dl":null,"odo":null,"drst":null,"oday":"2019-10-
29","jrn":76406215,"line":"8","start":"19:55","loc":"MAN","stop":1431604,"route":"31M1","
occu":0,"seq":1}}
offset = 1118, key =
/hfp/v2/journey/ongoing/vp/metro/0050/00157/31M1/2/Matinkylä/19:55/1431604/3/60;25/20/04/
54, value = {"VP":{"desi":"M1","dir":"2","oper":50,"veh":157,"tst":"2019-10-
29T18:03:24.098Z","tsi":1572372204,"spd":12.95,"hdg":232,"lat":60.20571237,"long":25.0444
6321,"acc":null,"dl":null,"odo":null,"drst":null,"oday":"2019-10-
29","jrn":76406264,"line":"8","start":"19:55","loc":"MAN","stop":1431604,"route":"31M1","
occu":0,"seq":2}}
offset = 1119, key = /hfp/v2/journey/ongoing/vp/tram/0040/00408/1009/1/Pasila
as./19:46/1111431/4/60;24/19/74/99, value =
{"VP":{"desi":"9","dir":"1","oper":40,"veh":408,"tst":"2019-10-
29T18:03:29.703Z","tsi":1572372209,"spd":0.60,"hdg":357,"lat":60.179503,"long":24.949977,
"acc":0.31,"dl":0,"odo":3684,"drst":0,"oday":"2019-10-
29","jrn":1692,"line":39,"start":"19:46","loc":"GPS","stop":1111428,"route":"1009","occu"
:0}}
...
```

As you can see, we get our vehicle positions back, that the producer feeds into the topic.

It can happen that a (mostly transient) error at the MQTT source is observed when running the producer. You will see an error message starting as follows: `Exception in thread "main" MqttException (0) - java.net.UnknownHostException: mqtt.hsl.fi`... and the producer stops.
In this case just restart the producer.

7. Stop the application by clicking the **stop** button in the debugging toolbar.

OPTIONAL: Creating a Docker Image for the Consumer

To prepare for a later lab, we want to create a Docker image containing our consumer, so we can easily run it as a part of our distributed application.

1. In VS Code open the file `Dockerfile` in the project folder (`~/confluent-dev/labs/create-consumer-java`) and observe the steps needed to containerize our Java application. Notice that the Dockerfile is organized as a multi-step manifest, to reduce the size of the final image.
2. In the terminal, from within the project folder execute the following command to build the image:

```
$ cd ~/confluent-dev/labs/create-consumer-java
$ docker image build -t cnfltraining/vp-consumer:v2 .
```

Don't forget the period (`.`) at the end of the above command! It instructs the Docker builder to use the current directory as the context.

The first time you build the image it takes a while, mainly because the builder needs to restore the Maven packages. Please be patient.

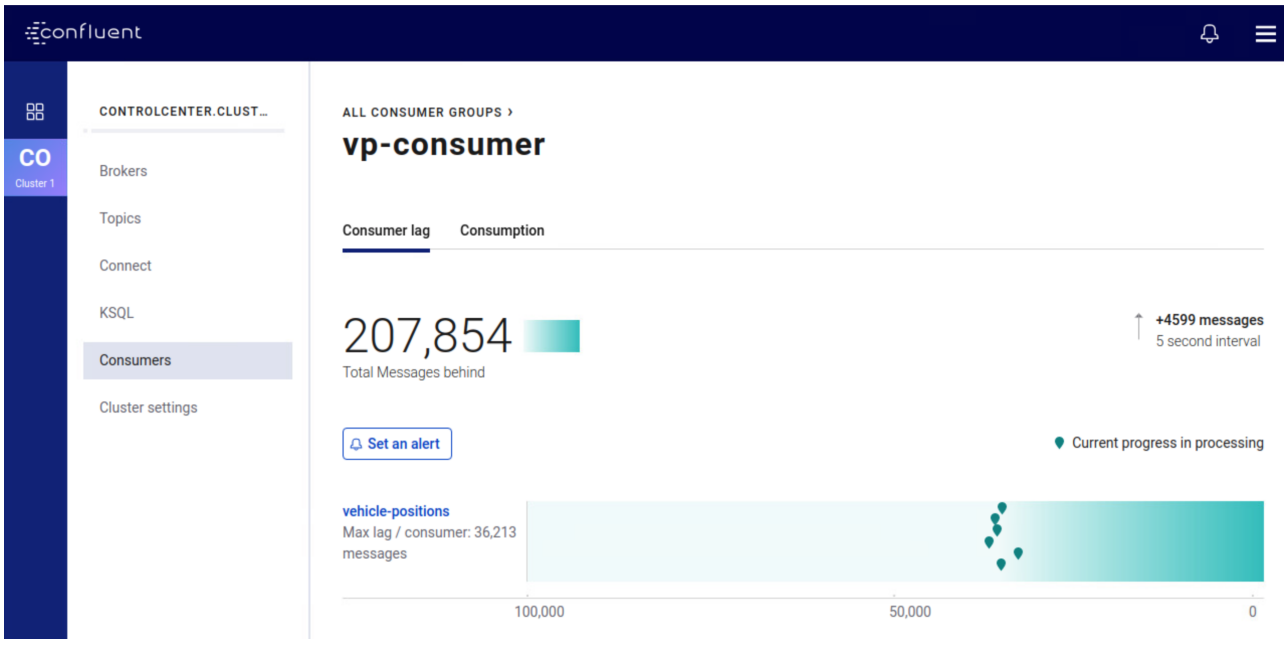
3. Test the new image by running a container from it:

```
$ docker container run --rm -it \
    --name vp_consumer \
    --net labs_confluent \
    cnfltraining/vp-consumer:v2
```

You should see an output like this (shortened):

```
* Starting VP Consumer *
...
offset = 1128, key =
/hfp/v2/journey/ongoing/vp/metro/0050/00149/31M2/1/Mellunmäki/19:38/1431601/3/60;25/10/93
/40, value = {"VP":{"desi":"M2","dir":"1","oper":50,"veh":149,"tst":"2019-10-
29T18:02:51.785Z","tsi":1572372171,"spd":12.23,"hdg":66,"lat":60.19441891,"long":25.03009
729,"acc":null,"dl":null,"odo":null,"drst":null,"oday":"2019-10-
29","jrn":76406860,"line":"17","start":"19:38","loc":"MAN","stop":1431601,"route":"31M2",
"occu":0,"seq":1}}
offset = 1129, key =
/hfp/v2/journey/ongoing/vp/metro/0050/00175/31M1/2/Matinkylä/19:55/1431604/3/60;25/20/04/
54, value = {"VP":{"desi":"M1","dir":"2","oper":50,"veh":175,"tst":"2019-10-
29T18:03:28.724Z","tsi":1572372208,"spd":9.18,"hdg":232,"lat":60.20571237,"long":25.04446
321,"acc":null,"dl":null,"odo":null,"drst":null,"oday":"2019-10-
29","jrn":76406215,"line":"8","start":"19:55","loc":"MAN","stop":1431604,"route":"31M1","
occu":0,"seq":1}}
offset = 1130, key =
/hfp/v2/journey/ongoing/vp/bus/0022/01015/2532/2/Matinkylä/19:52/2131246/4/60;24/27/16/12
, value = {"VP":{"desi":"532","dir":"2","oper":22,"veh":1015,"tst":"2019-10-
29T18:03:35.262Z","tsi":1572372215,"spd":7.75,"hdg":226,"lat":60.211067,"long":24.762192,
"acc":-0.76,"dl":-46,"odo":5196,"drst":0,"oday":"2019-10-
29","jrn":439,"line":865,"start":"19:52","loc":"GPS","stop":null,"route":"2532","occu":0}
}
...
```

4. Open Confluent Control Center (http://localhost:9021):
   a. select cluster **Cluster 1**
   b. select the **Consumers** tab
   c. select the consumer group `vp-consumer`
   d. observe how the consumer lag is behaving:



5. End the consumer by pressing `CTRL-C`.

The complete solution can be found at `~/confluent-dev/solution/create-consumer-java`.

OPTIONAL: Create the same Consumer in Python

In this optional exercise you are supposed to create a consumer with the same functionality as above, but in Python. Use the Confluent supported Kafka client library for Python. You can find examples how to use this library here: https://github.com/confluentinc/confluent-kafka-python.

1. Start by moving to the project folder

```
$ cd ~/confluent-dev/labs/create-consumer-python
```

2. Make sure your dependencies are installed:

```
$ pip3 install -r requirements.txt
```

3. Open the project in VS Code:

```
$ code .
```

4. Add code to the file `main.py`

Once you have written your consumer you can build a Docker image from it as follows:

```
$ docker image build -t cnfltraining/vp-consumer-python:v2 .
```

You can then (if you want) run the consume as a container as follows:

```
$ docker container run --rm -it \
   --name vp-consumer-python \
   --net labs_confluent \
   cnfltraining/vp-consumer-python:v2
```

The complete solution can be found at `~/confluent-dev/solution/create-consumer-python`.

OPTIONAL: Create the same Consumer in .NET C#

In this optional exercise you are supposed to create a consumer with the same functionality as above, but in C# on .NET. Use the Confluent supported Kafka client library for .NET. You can find examples how to use this library here: https://github.com/confluentinc/confluent-kafka-dotnet.

1. Start by moving to the project folder

```
$ cd ~/confluent-dev/labs/create-consumer-net
```

2. Restore the project (which downloads all needed dependencies) by running:

```
$ dotnet restore
```

3. Open VS Code with:

```
$ code .
```

4. Add your code to the file `Program.cs`.

Once you have written your consumer you can build a Docker image from it as follows:

```
$ docker image build -t cnfltraining/vp-consumer-net:v2 .
```

You can then (if you want) run the consume as a container as follows:

```
$ docker container run --rm -it \
   --name vp-consumer-net \
   --net labs_confluent \
   cnfltraining/vp-consumer-net:v2
```

The complete solution can be found at `~/confluent-dev/solution/create-consumer-dotnet`.

Cleanup

1. Stop the producer with this command:

```
$ docker kill vp-producer
```

2. Execute the following command to completely cleanup your environment:

```
$ docker-compose down -v
```

Conclusion

In this exercise you have created a Kafka consumer in Java that consumes data from a topic called `vehicle-positions`. The records in that topic are produced by a Kafka producer that ingests live data of the **High-frequency Positioning** service of **digitransit**. We had created the producer in the last lab **Creating a Producer**.

STOP HERE. THIS IS THE END OF THE EXERCISE.

Lab 05a Accessing Previous Data in Java

Accessing Previous Data in Java

In this Hands-On Exercise, you will modify the existing consumer such as that it accesses data starting from the beginning of the topic each time it launches.

Prerequisites

1. Navigate to the project folder:

```
$ cd ~/confluent-dev/labs/prev-data-java
```

2. Run the Kafka cluster, including Confluent Control Center:

```
$ docker-compose up -d
```

3. Create the topic `vehicle-positions` with 6 partitions on Kafka with this command:

```
$ kafka-topics \
    --create \
    --bootstrap-server kafka:9092 \
    --partitions 6 \
    --replication-factor 1 \
    --topic vehicle-positions
```


4. Run a Docker container with the Kafka producer from the lab **Creating a Kafka Producer**:

```
$ docker container run --rm -d \
    --name vp-producer \
    --hostname vp-producer \
    --net labs_confluent \
    cnfltraining/vp-producer:v2
```

If you were not able to finish the lab **Creating a Kafka Producer** then don't worry. The necessary image `cnfltraining/vp-producer:v2` will just be downloaded from our Confluent Training org on Docker Hub.

If you encounter problems with the producer, e.g. the MQTT datasource that the producer is relying on is down, then please use the Docker image `cnfltraining/vp-producer-fallback:v2` instead.

5. Open the project in Visual Studio Code:

```
$ code .
```

Writing the Consumer

1. Open the file `src/main/java/clients/PrevDataConsumer.java` and note that it already contains the nearly the same code that you were using in the lab **Creating a Kafka Consumer**.
2. Modify this code such that it reads all data from the topic `vehicle-positions` each time it starts:
   a. Right after the line where we instantiate the consumer, add a definition for a `ConsumerRebalanceListener` as follows:

```
ConsumerRebalanceListener listener = new ConsumerRebalanceListener() {
  @Override
  public void onPartitionsRevoked(Collection<TopicPartition> partitions) {
              // nothing to do...
  }

  @Override
  public void onPartitionsAssigned(Collection<TopicPartition> partitions) {
    consumer.seekToBeginning(partitions);
  }
};
```

   b. Then modify the line where we subscribe the listener to the topic `vehicle-positions` to look as follows:

```
consumer.subscribe(Arrays.asList("vehicle-positions"), listener);
```

   Note how we pass the listener object as the second argument to the `subscribe` method.

3. Run the application select the menu **Debug → Start Debugging** in VS Code. The output of the consumer should look similar to this (shortened):

```
*** Starting Prev Data Consumer ***
partition = 5, offset = 0, key =
/hfp/v2/journey/ongoing/vp/bus/0018/00288/1059/2/Sompasaari/19:43/1174128/4/60;24/19/92/9
7, value = {"VP":{"desi":"59","dir":"2","oper":18,"veh":288,"tst":"2019-10-
29T17:59:46.695Z","tsi":1572371986,"spd":9.10,"hdg":166,"lat":60.199416,"long":24.927401,
"acc":-0.65,"dl":-148,"odo":5002,"drst":0,"oday":"2019-10-
29","jrn":1281,"line":77,"start":"19:43","loc":"GPS","stop":null,"route":"1059","occu":0}
}
partition = 5, offset = 1, key =
/hfp/v2/journey/ongoing/vp/metro/0050/00314/31M2/2/Tapiola/16:57/1431602/3/60;25/10/93/86
, value = {"VP":{"desi":"M2","dir":"2","oper":50,"veh":314,"tst":"2019-10-
29T17:59:46.749Z","tsi":1572371986,"spd":19.69,"hdg":229,"lat":60.19865403,"long":25.0361
0106,"acc":null,"dl":null,"odo":null,"drst":null,"oday":"2019-10-
29","jrn":76405115,"line":"2","start":"16:57","loc":"MAN","stop":null,"route":"31M2","occ
u":0,"seq":1}}
...
```

Notice how we start from offset 0!

4. Stop the consumer by pressing the **STOP** button on the debug toolbar in VS Code.
5. Run the consumer again and verify that it starts at offset zero, as expected.
6. Stop the consumer by pressing the **STOP** button on the debug toolbar in VS Code.
7. Exit Visual Studio Code

The complete solution can be found at `~/confluent-dev/solution/prev-data-java`.

OPTIONAL: Accessing Previous Data In C#/.NET

If you have more time and feel up to the challenge then please write a consumer in .NET C# having the same functionality than the Java consumer you just wrote.

If you have skipped the Java version above and just want to do the .NET version then please make sure you have followed step 2, 3 and 4 of the **Prerequisites** section above, such as that you have a running cluster and a producer generating data into the topic `vehicle-positions`.

1. Navigate to the project folder:

```
$ cd ~/confluent-dev/labs/prev-data-net
```

2. Open the project in VS Code:

```
$ code .
```

3. When prompted click **Restore** to install NuGet dependencies.

4. Locate the file `app.csproj`. Open it and notice that we're using the `Confluent.Kafka` library:

```
...
  <ItemGroup>
    <PackageReference Include="Confluent.Kafka" Version="1.1.0" />
  </ItemGroup>
...
```

5. Open the file `Program.cs` located in the same folder. It should have this content:

```
using System;
using System.Linq;
using System.Threading;
using Confluent.Kafka;

namespace app {
    class Program {
        static void Main (string[] args) {
            Console.WriteLine ("Starting .NET Consumer!");
            var conf = new ConsumerConfig {
                    GroupId = "vp-consumer-group-net",
                    BootstrapServers = "kafka:9092",
                    AutoCommitIntervalMs = 5000,
                    AutoOffsetReset = AutoOffsetReset.Earliest
                };

            using (var consumer = new ConsumerBuilder<string, string> (conf)
                .Build())
            {
                consumer.Subscribe("vehicle-positions");

                CancellationTokenSource cts = new CancellationTokenSource();
                Console.CancelKeyPress += (_, e) => {
                    e.Cancel = true; // prevent the process from terminating.
                    cts.Cancel();
                };

                try
                {
                    while (true)
                    {
                        try
                        {
                            var cr = consumer.Consume(cts.Token);
                            Console.WriteLine($"Consumed message '{cr.Value}' at: '{cr
.TopicPartitionOffset}'.");
                        }
                        catch (ConsumeException e)
                        {
                            Console.WriteLine($"Error occurred: {e.Error.Reason}");
                        }
                    }
                }
                catch (OperationCanceledException)
                {
                    // Ensure the consumer leaves the group cleanly
                    // and final offsets are committed.
                    consumer.Close();
                }
            }
        }
    }
}
```


Here we're defining the configuration for the Kafka Consumer in the variable `conf` and the use it to create an

instance of `Consumer` with key and value of type `string`.

6. Now we're adding code to the `using` block to react on the partition assignment event. Modify the code that builds the consumer as follows:

```
using (var consumer = new ConsumerBuilder<string, string> (conf)
    .SetPartitionsAssignedHandler((c, partitions) =>
    {
        Console.WriteLine("Resetting all partitions to offset 0.");
        return partitions.Select(tp =>
            new TopicPartitionOffset(tp, new Offset(0)));
    })
    .Build())
{
  ...
};
```

In the event handler defined in the method `SetPartitionsAssignedHandler` we have the code that always resets the offset of all partitions of our selected topic to the **beginning** (=0) as required in this particular exercise.

7. In VS Code click the menu **Debug → Start Debugging** to start debugging, which in my case generates this output:


```
...
Starting .NET Consumer!
...
Resetting all partitions to offset 0.
Consumed message '{"VP":{"desi":"39","dir":"1","oper":12,"veh":1401,"tst":"2019-10-
29T17:59:46.656Z","tsi":1572371986,"spd":7.64,"hdg":335,"lat":60.226664,"long":24.853910,
"acc":-0.83,"dl":-314,"odo":null,"drst":null,"oday":"2019-10-
29","jrn":858,"line":60,"start":"19:28","loc":"GPS","stop":null,"route":"1039","occu":0}}
' at: 'vehicle-positions [[0]] @0'.
Consumed message '{"VP":{"desi":"17","dir":"1","oper":12,"veh":934,"tst":"2019-10-
29T17:59:46.602Z","tsi":1572371986,"spd":6.59,"hdg":265,"lat":60.158319,"long":24.946851,
"acc":0.75,"dl":-5,"odo":4731,"drst":0,"oday":"2019-10-
29","jrn":123,"line":46,"start":"19:43","loc":"GPS","stop":null,"route":"1017","occu":0}}
' at: 'vehicle-positions [[0]] @1'.
...
```

You can see that the first item has offset 0 (`'vehicle-positions [[0]] @0'`). This is an indication that the consumer group started consuming at the earliest point.

8. In the terminal window where you started the producer, stop it with:

```
$ docker container kill vp-producer
```

9. Wait until the consumer has caught up. Then stop the consumer by end debugging in VS Code.

10. Start the consumer again by clicking **Debug → Start Debugging** in VS Code and observe that it starts from scratch again.
11. Stop the consumer by stopping debugging in VS Code.
12. Exit Visual Studio Code

The complete solution can be found at `~/confluent-dev/labs/solution/prev-data-dotnet`.

Cleanup

1. Execute the following commands to completely cleanup your environment:

```
$ docker container kill vp-producer
$ docker-compose down -v
```

Conclusion

In this exercise you have modified our Vehicle Positions consumer such as that it always reads data from the very beginning of the topic `vehicle-positions`, no matter how many times we restart it. Optionally you had the opportunity to write the consumer in .NET C#.

STOP HERE. THIS IS THE END OF THE EXERCISE.

Lab 05b Managing Consumer Offsets in Code - Java

Managing Consumer Offsets in Code - Java

In this exercise you will write a consumer which processes records from the topic `vehicle-positions` and manually manages offsets by writing them to files on disk; it should write the offset after each message. To 'process' the message, just display it to the console. When the consumer starts, it should seek to the correct offset. Modify your Consumer so that it halts randomly during processing so you can confirm that it performs correctly when restarted.

The motivation for this could be that you want your consumer to support exactly once semantic (EOS). By storing the offset with the data in the same transaction in an external DB you could achieve this.

Prerequisites

1. Navigate to the project folder:

```
$ cd ~/confluent-dev/labs/manual-offset-java
```

2. Run the Kafka cluster, including Confluent Control Center:

```
$ docker-compose up -d
```


3. Create the topic `vehicle-positions` with 6 partitions on Kafka with this command:

```
$ kafka-topics \
    --create \
    --bootstrap-server kafka:9092 \
    --partitions 6 \
    --replication-factor 1 \
    --topic vehicle-positions
```

If the topic already exists then you get an error:
`Error while executing topic command : Topic vehicle-positions already exists....`

4. Run a Docker container with the Kafka producer from the lab **Creating a Kafka Producer**:

```
$ docker container run --rm -d \
    --name vp-producer \
    --hostname vp-producer \
    --net labs_confluent \
    cnfltraining/vp-producer:v2
```

If you were not able to finish the lab **Creating a Kafka Producer** then don't worry. The necessary image `cnfltraining/vp-producer:v2` will just be downloaded from our Confluent Training org on Docker Hub.

If you encounter problems with the producer, e.g. the MQTT datasource that the producer is relying on is down, then please use the Docker image `cnfltraining/vp-producer-fallback:v2` instead.

5. Open the project in Visual Studio Code:

```
$ code .
```

Writing the Consumer

1. Open the file `src/main/java/clients/VehiclePositionConsumer.java` and note that it contains well known code as we have first encountered in the lab **Creating a Kafka Consumer**.
2. First we add a static variable to the class:

```
final static String OFFSET_FILE_PREFIX = "./offsets/offset_";
```

3. Next we add a private static function to create a `ConsumerRebalanceListener`. This listener is used to initialize the consumer with the offsets read from the file. Add this code right after the `main` function:

```
private static ConsumerRebalanceListener createListener(KafkaConsumer<String, String>
consumer){
    return new ConsumerRebalanceListener() {
        @Override
        public void onPartitionsRevoked(Collection<TopicPartition> partitions) {
            // nothing to do...
        }

        @Override
        public void onPartitionsAssigned(Collection<TopicPartition> partitions) {
            for (TopicPartition partition : partitions) {
                try{
                    if (Files.exists(Paths.get(OFFSET_FILE_PREFIX + partition.
partition()))) {
                        long offset = Long
                            .parseLong(Files.readAllLines(Paths.get(OFFSET_FILE_PREFIX +
partition.partition()),
                                                Charset.defaultCharset()).get(0));
                        consumer.seek(partition, offset);
                    }
                } catch(IOException e) {
                    System.out.printf("ERR: Could not read offset from file.\n");
                }
            }
        }
    };
}
```


4. Now add the following line right after the instantiation of the consumer in the main method:

```
ConsumerRebalanceListener listener = createListener(consumer);
```

5. Modify the line where the consumer subscribes to the topic `vehicle-positions` and add the `listener` object as second parameter:

```
consumer.subscribe(Arrays.asList("vehicle-positions"), listener);
```

6. Modify the `for` loop such as that it not only outputs each record to the console but also writes the current offset to the respective file:

```
for (ConsumerRecord<String, String> record : records) {
    System.out.printf("offset = %d, key = %s, value = %s\n",
            record.offset(), record.key(), record.value());
    Files.write(Paths.get(OFFSET_FILE_PREFIX + record.partition()),
            Long.valueOf(record.offset() + 1).toString().getBytes());
}
```

7. Run the application by clicking **Debug → Start Debugging** in VS Code. The output should look similar to this (shortened):

```
* Starting VP Consumer *
partition = 5, offset = 0, key =
/hfp/v2/journey/ongoing/vp/bus/0018/00288/1059/2/Sompasaari/19:43/1174128/4/60;24/19/92/9
7, value = {"VP":{"desi":"59","dir":"2","oper":18,"veh":288,"tst":"2019-10-
29T17:59:46.695Z","tsi":1572371986,"spd":9.10,"hdg":166,"lat":60.199416,"long":24.927401,
"acc":-0.65,"dl":-148,"odo":5002,"drst":0,"oday":"2019-10-
29","jrn":1281,"line":77,"start":"19:43","loc":"GPS","stop":null,"route":"1059","occu":0}
}
partition = 5, offset = 1, key =
/hfp/v2/journey/ongoing/vp/metro/0050/00314/31M2/2/Tapiola/16:57/1431602/3/60;25/10/93/86
, value = {"VP":{"desi":"M2","dir":"2","oper":50,"veh":314,"tst":"2019-10-
29T17:59:46.749Z","tsi":1572371986,"spd":19.69,"hdg":229,"lat":60.19865403,"long":25.0361
0106,"acc":null,"dl":null,"odo":null,"drst":null,"oday":"2019-10-
29","jrn":76405115,"line":"2","start":"16:57","loc":"MAN","stop":null,"route":"31M2","occ
u":0,"seq":1}}
...
```

8. Notice that in the subfolder `offsets` of the project folder there are now 6 files `offset_0` to `offset_5` which if you view their content contain the current offset of the respective topic partition.
9. Quit the consumer by stop debugging in VS Code.
10. Restart the consumer. It should continue from the offset found in the files located in folder `offsets`.

The complete solution can be found at `~/confluent-dev/solution/manual-offset-java`.

OPTIONAL: Managing Consumer Offsets in Code - C#/.NET

If you have more time and feel up to the challenge then please write a consumer in .NET C# having the same functionality as the Java consumer you just wrote.

1. Navigate to the project folder:

```
$ cd ~/confluent-dev/labs/manual-offset-net
```

2. Open the project in VS Code:

```
$ code .
```

3. When prompted click **Restore** to install NuGet dependencies.
4. Open the file `Program.cs` located in the same folder. It should look familiar to you, since it is the same code that we used in the lab about accessing previous data.
5. First add a class variable `OFFSET_FILE_PREFIX` to the code:

```
static string OFFSET_FILE_PREFIX = "./offsets/offset_";
```

6. In the `try-catch` block where we consume records add this code snippet to save the current offset of the respective partition into a file called `offset_X` (where X is the partition number):

```
File.WriteAllText(Path.GetFullPath(
    OFFSET_FILE_PREFIX + cr.TopicPartition.Partition.Value),
    cr.TopicPartitionOffset.Offset.Value.ToString());
```

7. In the partition assignment handler

   a. Change the message output to the console from `Resetting all partitions to offset 0.` to `Resetting all partitions to offset found in file.`

   b. add code to read the offsets from disk if available and assign it to the respective partition:

```
return partitions.Select(tp => {
    var path = Path.GetFullPath(OFFSET_FILE_PREFIX + tp.Partition.Value);
    long offset = tp.Partition.Value;
    if(File.Exists(path)){
        offset = long.Parse(File.ReadAllText(path));
    }
    return new TopicPartitionOffset(tp, new Offset(offset));
});
```

8. In VS Code click the menu **Debug → Start Debugging** to start debugging, which in my case generates this output:

```
...
Resetting all partitions to offset found in file.
Consumed message '{"VP":{"desi":"39","dir":"1","oper":12,"veh":1401,"tst":"2019-10-
29T17:59:46.656Z","tsi":1572371986,"spd":7.64,"hdg":335,"lat":60.226664,"long":24.853910,
"acc":-0.83,"dl":-314,"odo":null,"drst":null,"oday":"2019-10-
29","jrn":858,"line":60,"start":"19:28","loc":"GPS","stop":null,"route":"1039","occu":0}}
' at: 'vehicle-positions [[0]] @0'.
Consumed message '{"VP":{"desi":"17","dir":"1","oper":12,"veh":934,"tst":"2019-10-
29T17:59:46.602Z","tsi":1572371986,"spd":6.59,"hdg":265,"lat":60.158319,"long":24.946851,
"acc":0.75,"dl":-5,"odo":4731,"drst":0,"oday":"2019-10-
29","jrn":123,"line":46,"start":"19:43","loc":"GPS","stop":null,"route":"1017","occu":0}}
' at: 'vehicle-positions [[0]] @1'.
...
```

9. Make sure in the folder `offsets` in the project folder you are seeing the files `offset_0` to `offset_5`. Also make sure they contain the current offset of the respective partition.
10. Stop the consumer by ending debugging in VS Code
11. Start the consumer again and make sure the offsets are correctly read from the disk.
12. Stop the consumer by ending debugging in VS Code

Cleanup

1. Stop the producer with:

```
$ docker kill vp-producer
```

2. Execute the following command to completely cleanup your environment:

```
$ docker-compose down -v
```

Conclusion

In this exercise you have modified the Vehicle Positions consumer such that it always writes its current offset to a file. Upon start the consumer reads the last processed offset from that file and seeks in the topic partition to that position before starting to process messages.

STOP HERE. THIS IS THE END OF THE EXERCISE.

Lab 06 Using Kafka with Avro

Using Kafka with Avro

Once again we are working with MQTT data from: https://digitransit.fi/en/developers/apis/4-realtime-api/vehicle-positions/

Prerequisites

1. Navigate to the project folder:

```
$ cd ~/confluent-dev/labs/avro-java
```

2. Run the Kafka cluster, including Confluent Control Center:

```
$ docker-compose up -d
```

3. Create the topic `vehicle-positions-avro` with 6 partitions on Kafka with this command:

```
$ kafka-topics \
    --create \
    --bootstrap-server kafka:9092 \
    --partitions 6 \
    --replication-factor 1 \
    --topic vehicle-positions-avro
```


Writing a Avro Producer

1. Navigate to the producer folder:

```
$ cd producer
```

2. `build.gradle` applies an Avro code generation plugin. Use this to generate the POJOs (Plain Old Java Objects/classes) from the Avro schemas.

```
$ gradle build
```

3. Open the producer project in VS Code:

```
$ code .
```

4. Locate and analyze the two classes `VehiclePositionProducer` and `Subscriber`. They contain the same code you were using in the lab **Creating a Kafka Producer in Java**.
5. We need now to change the latter class to use the Avro data format for the record values, instead of just plain text.
6. First open the file `build.gradle` and notice that we have added:
   - 3 dependencies:

     ```
     compile group: 'org.apache.avro', name: 'avro', version: '1.8.2'
     compile group: 'org.apache.avro', name: 'avro-tools', version: '1.8.2'
     compile group: 'io.confluent', name: 'kafka-avro-serializer', version: '5.3.0'
     ```

   - and an Avro plugin:
     `com.commercehub.gradle.plugin.avro`
     which includes a task `generateAvroJava` to generate POJOs (Plain Old Java Objects/classes) from any Avro schemas in the project
7. Locate the folder `src/main/avro` and in it the two files `position_key.avsc` and `position_value.avsc`. Analyze the two files to discover how a typical Avro schema is defined.
8. Now let's change the configuration of the producer in the class `VehiclePositionProducer` as follows:

   ```
   Properties settings = new Properties();
   settings.put(ProducerConfig.CLIENT_ID_CONFIG, "vp-producer-avro");
   settings.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "kafka:9092");
   settings.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, KafkaAvroSerializer.class);
   settings.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, KafkaAvroSerializer.class);
   settings.put(KafkaAvroSerializerConfig.SCHEMA_REGISTRY_URL_CONFIG, "http://schema-
   registry:8081");
   ```

   Notice how we're using the `ProducerConfig` helper class to avoid using strings as keys, that might lead to subtle errors, such as typos. Also notice how we define the key and value serializers to be of type Avro, and finally how we define the link to our Schema Registry.

9. Next, we modify the way how we instantiate the producer. Instead of being generic in `String` for both key and value it is now generic in `PositionKey` and `PositionValue`:

   ```
   final KafkaProducer<PositionKey, PositionValue> producer = new KafkaProducer<>(settings);
   ```

10. Now we update the `Subscriber` class:
    a. Change the value of the instance variable `kafka_topic` from `vehicle-positions` to `vehicle-positions-avro`
    b. Modify the constructor and the instance variable as follows:

```
private KafkaProducer<PositionKey, PositionValue> producer;

public Subscriber(KafkaProducer<PositionKey, PositionValue> producer) {
    this.producer = producer;
}
```

c. Modify the method `messageArrived` like so:

```
public void messageArrived(String topic, MqttMessage message) throws MqttException {
    try {
        System.out.println(String.format("[%s] %s",
                topic, new String(message.getPayload())));
        final PositionKey key = new PositionKey(topic);
        final PositionValue value = getPositionValue(message.getPayload());
        final ProducerRecord<PositionKey, PositionValue> record =
                new ProducerRecord<>(this.kafka_topic, key, value);
        producer.send(record);
    } catch (Exception e) {
        //TODO: handle exception
    }
}
```

d. Add the private helper method `getPositionValue` which maps the payload of the MQTT record received to an instance of `PositionValue`:

```
private PositionValue getPositionValue(byte[] payload) throws IOException {
    ObjectMapper mapper = new ObjectMapper();
    String json = new String(payload);
    VehiclePosition pos =  mapper.readValue(json, VehiclePosition.class);
    VehicleValues vv = pos.VP;

    return new PositionValue(vv.desi, vv.dir, vv.oper, vv.veh, vv.tst,
        vv.tsi, vv.spd, vv.hdg, vv.lat, vv.longitude, vv.acc, vv.dl,
        vv.odo, vv.drst, vv.oday, vv.jrn, vv.line, vv.start, vv.loc,
        vv.stop, vv.route, vv.occu, vv.seq);
}
```

11. Add a file `VehiclePosition.java` to the `src/main/java/clients` folder with this content:

```
package clients;

import com.fasterxml.jackson.annotation.JsonProperty;

public class VehiclePosition {
    public VehiclePosition() {}

    @JsonProperty("VP")
    public VehicleValues VP;

    public class VehicleValues{
```

```
        @JsonProperty("desi")
        public String desi;
        @JsonProperty("dir")
        public String dir;
        @JsonProperty("oper")
        public int oper;
        @JsonProperty("veh")
        public int veh;
        @JsonProperty("tst")
        public String tst;
        @JsonProperty("tsi")
        public long tsi;
        @JsonProperty("spd")
        public Double spd;
        @JsonProperty("hdg")
        public int hdg;
        @JsonProperty("lat")
        public Double lat;
        @JsonProperty("long")
        public Double longitude;
        @JsonProperty("acc")
        public Double acc;
        @JsonProperty("dl")
        public int dl;
        @JsonProperty("odo")
        public int odo;
        @JsonProperty("drst")
        public int drst;
        @JsonProperty("oday")
        public String oday;
        @JsonProperty("jrn")
        public int jrn;
        @JsonProperty("line")
        public int line;
        @JsonProperty("start")
        public String start;
        @JsonProperty("loc")
        public String loc;
        @JsonProperty("stop")
        public String stop;
        @JsonProperty("route")
        public String route;
        @JsonProperty("occu")
        public int occu;
        @JsonProperty("seq")
        public int seq;
    }
}
```


`VehiclePosition` is a helper class that we use to convert the JSON string into a Java object.

12. Return to `Subscriber.java` and uncomment the import of the new `VehicleValues` class.

```
import clients.VehiclePosition.VehicleValues;
```

13. Run the application by selecting the menu **Debug** → **Start Debugging** in VS Code.

    You should see something along the line:

```
*** Starting VP Producer ***
...
[/hfp/v2/journey/ongoing/vp/tram/0040/00111/1002/1/Pasila
as./10:57/1171404/4/60;24/19/92/39]
{"VP":{"desi":"2","dir":"1","oper":40,"veh":111,"tst":"2019-10-
30T09:22:43.058Z","tsi":1572427363,"spd":5.72,"hdg":30,"lat":60.193076,"long":24.929415,"
acc":1.21,"dl":101,"odo":5823,"drst":0,"oday":"2019-10-
30","jrn":732,"line":30,"start":"10:57","loc":"GPS","stop":1171458,"route":"1002","occu":
0}}
[/hfp/v2/journey/ongoing/vp/bus/0047/00805/1819/2/Vuosaari(M)/11:20/1543100/0////]
{"VP":{"desi":"819","dir":"2","oper":47,"veh":805,"tst":"2019-10-
30T09:22:43.057Z","tsi":1572427363,"spd":null,"hdg":null,"lat":null,"long":null,"acc":nul
l,"dl":-120,"odo":null,"drst":null,"oday":"2019-10-
30","jrn":55,"line":599,"start":"11:20","loc":"ODO","stop":1543100,"route":"1819","occu":
0}}
..
```

14. Open Confluent Control Center (http://localhost:9021):
    a. select cluster **Cluster 1**
    b. select the **Topics** tab
    c. in the list of topics select the topic `vehicle-positions-avro`
    d. switch to the **Messages** tab and observe the inflowing (Avro) records
15. End the producer by clicking the **Stop** button on the debugging toolbar in VS Code.

Question: We have not defined proper exception handling in the `Subscriber.java` class. How would you ideally handle those exceptions before you put the application in production? Discuss with your peers.

Writing an Avro Consumer

1. Navigate to the consumer project folder:

```
$ cd ~/confluent-dev/labs/avro-java/consumer
```

2. Run a gradle build to generate POJOs from the Avro schemas, then launch VS Code.

```
$ gradle build
$ code .
```

   The code is the same we used in the lab **Creating a Kafka Consumer**.

3. First open the file `build.gradle` and notice that we have (similarly to what we did for the producer) added:

- 3 dependencies:

```
compile group: 'org.apache.avro', name: 'avro', version: '1.8.2'
compile group: 'org.apache.avro', name: 'avro-tools', version: '1.8.2'
compile group: 'io.confluent', name: 'kafka-avro-serializer', version: '5.3.0'
```

- and an Avro plugin:
  `com.commercehub.gradle.plugin.avro`
  which includes a task `generateAvroJava` to generate POJOs (Plain Old Java Objects/classes) from any Avro schemas in the project

4. Notice the two Avro schemas in the folder `src/main/avro`. They will be converted into POJOs of type `PositionKey` and `PositionValue` during compile time.
5. Let's now modify the consumer class `VehiclePositionConsumer`:
   a. First modify the imports section to look like this:

```
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;
import org.apache.kafka.clients.consumer.ConsumerConfig;
import io.confluent.kafka.serializers.KafkaAvroDeserializer;
import io.confluent.kafka.serializers.KafkaAvroDeserializerConfig;
import solution.model.PositionKey;
import solution.model.PositionValue;
```


   b. Then modify the configuration of the consumer as follows:

```
Properties settings = new Properties();
settings.put(ConsumerConfig.GROUP_ID_CONFIG, "vp-consumer-avro");
settings.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "kafka:9092");
settings.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest");
settings.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, KafkaAvroDeserializer.
class);
settings.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG, KafkaAvroDeserializer
.class);
settings.put(KafkaAvroDeserializerConfig.SPECIFIC_AVRO_READER_CONFIG, "true");
settings.put(KafkaAvroDeserializerConfig.SCHEMA_REGISTRY_URL_CONFIG, "http://schema-
registry:8081");
```

the second last line is specifically important. It easily gets forgotten. If you omit it, the record key and value will not be converted into the specific types `PositionKey` and `PositionValue`!

   c. We also need to change the definition of the `KafkaConsumer`:

```
KafkaConsumer<PositionKey, PositionValue> consumer =
    new KafkaConsumer<>(settings);
```

d. Then we need to change the name of the topic we subscribe to, to `vehicle-positions-avro`

e. Finally adjust the `while` loop to work with the correct types:

```
while (true) {
    ConsumerRecords<PositionKey, PositionValue> records = consumer.poll(Duration
.ofMillis(100));
    for (ConsumerRecord<PositionKey, PositionValue> record : records)
        System.out.printf("offset = %d, key = %s, value = %s\n",
            record.offset(), record.key().toString(), record.value().toString());
}
```

6. Run the consumer by selecting the menu **Debug** → **Start Debugging** in VS Code.

   You should see something like this:

```
*** Starting VP Consumer Avro ***
offset = 0, key = {"topic":
"/hfp/v2/journey/ongoing/vp/bus/0022/01123/2321/1/Vanhakartano/11:20/1130206/4/60;24/19/7
3/48"}, value = {"desi": "321", "dir": "1", "oper": 22, "veh": 1123, "tst": "2019-10-
30T09:22:38.414Z", "tsi": 1572427358, "spd": 4.77, "hdg": 318, "lat": 60.174515, "long":
24.938578, "acc": -0.73, "dl": -142, "odo": 583, "drst": 0, "oday": "2019-10-30", "jrn":
54, "line": 621, "start": "11:20", "loc": "GPS", "stop": null, "route": "2321", "occu":
0, "seq": 0}
offset = 1, key = {"topic":
"/hfp/v2/journey/ongoing/vp/tram/0040/00459/1006/2/Hietalahti/11:02/1020460/5/60;24/19/75
/80"}, value = {"desi": "6", "dir": "2", "oper": 40, "veh": 459, "tst": "2019-10-
30T09:22:38.419Z", "tsi": 1572427358, "spd": 0.0, "hdg": 174, "lat": 60.178511, "long":
24.950223, "acc": -0.2, "dl": -180, "odo": 4222, "drst": 0, "oday": "2019-10-30", "jrn":
2546, "line": 34, "start": "11:02", "loc": "GPS", "stop": "1111429", "route": "1006",
"occu": 0, "seq": 0}
...
```

7. End the consumer by clicking the **Stop** button on the debugging toolbar in VS Code.

OPTIONAL: Observe the Consumer Lag

1. Run both, the Avro producer and consumer.
2. In Confluent Control Center monitor the **Consumer lag** of the consumer group `vp-consumer-avro`
   a. Navigate to http://localhost:9021
   b. Select the cluster **Cluster 1**
   c. Select the tab **Consumers**
   d. From the list of consumers select `vp-consumer-avro`

e. Observe the consumer lag

3. Play with the consumer and producer by temporarily stopping and restarting them and observe the behavior of the consumer lag.

OPTIONAL: Create a new version of the schema

Here we are going to create a new schema using the REST API of the Schema Registry. We then register a version 2 of the same schema that should be BACKWARD compatible.

1. Navigate to the project folder:

```
$ cd ~/confluent-dev/labs/avro-java/schemas
```

2. Have a look at the content of the file `sample-schema-v1.json`. It contains a simple schema with one field
3. Register this new schema using `curl`:

```
$ curl -X POST \
    -H "Content-Type: application/vnd.schemaregistry.v1+json" \
    --data @sample-schema-v1.json \
    schema-registry:8081/subjects/sample/versions
```

4. Now have a look at the content of the file `sample-schema-v2.json`. It adds an optional field to the schema
5. Register this new version of the schema also using `curl`:

```
$ curl -X POST \
    -H "Content-Type: application/vnd.schemaregistry.v1+json" \
    --data @sample-schema-v2.json \
    schema-registry:8081/subjects/sample/versions
```

6. Let's see how many versions we have registered for subject `sample`:

```
$ curl -X GET \
    -H "Content-Type: application/vnd.schemaregistry.v1+json" \
    schema-registry:8081/subjects/sample/versions

[1,2]
```

As expected, we have version 1 and 2

7. Let's get the details of version 2:

```
$ curl -X GET \
    -H "Content-Type: application/vnd.schemaregistry.v1+json" \
    schema-registry:8081/subjects/sample/versions/2 | jq '.schema | fromjson'

...
{
    "type":"record",
    "name":"foo",
    "namespace":"solution.model",
    "fields":[
        {"name":"bar","type":"string"},
        {"name":"baz","type":"string","default":""}
    ]
}
```

8. We can delete e.g. version 2 as follows:

```
$ curl -X DELETE \
    -H "Content-Type: application/vnd.schemaregistry.v1+json" \
    schema-registry:8081/subjects/sample/versions/2
```

Cleanup

1. Execute the following command to completely cleanup your environment:

```
$ docker-compose down -v
```


Conclusion

In this exercise you have modified the Vehicle Positions producer such as that it uses Avro as a message format. You have then implemented a simple Kafka consumer, consuming the Avro records generated by the producer.

In production environments we recommend that client applications do not automatically register new schemas. This is one reason why in this lab we also learned how to create a schema via REST API.

STOP HERE. THIS IS THE END OF THE EXERCISE.

Lab 07a Using Kafka Connect

Using Kafka Connect

The goal of this lab is to use a Kafka Connect JDBC source connector to import data residing in PostgreSQL database into a topic in the Kafka cluster. This data can then be used to enrich our Vehicle Position data that we are getting from the **High-frequency Positioning** service of **digitransit**.

Prerequisites

1. Navigate to the project folder:

```
$ cd ~/confluent-dev/labs/using-connect
```

2. Run the Kafka cluster, including Confluent Control Center:

```
$ docker-compose up -d
```


Preparing the data in PostgreSQL

Now we are going to create an `operator` table in Postgres and fill it with data. For this we use the `psql` command line tool that is part of the `postgres` container that is running as part of our cluster.

1. Exec into the `postgres` container where you run `psql`:

```
$ docker-compose exec postgres psql -U postgres
psql (11.2)
Type "help" for help.

postgres=#
```

2. Create a table `operators` in PostgreSQL:

```
postgres=# CREATE TABLE operators(
    id int not null primary key,
    name varchar(50) not null
);

CREATE TABLE
```

3. Run the following statements to insert the operators in the corresponding table:

```
insert into operators(id, name) values(6, 'Oy Pohjolan Liikenne Ab');
insert into operators(id, name) values(12, 'Helsingin Bussiliikenne Oy');
insert into operators(id, name) values(17, 'Tammelundin Liikenne Oy');
insert into operators(id, name) values(18, 'Pohjolan Kaupunkiliikenne Oy');
insert into operators(id, name) values(19, 'Etelä-Suomen Linjaliikenne Oy');
insert into operators(id, name) values(20, 'Bus Travel Åbergin Linja Oy');
insert into operators(id, name) values(21, 'Bus Travel Oy Reissu Ruoti');
insert into operators(id, name) values(22, 'Nobina Finland Oy');
insert into operators(id, name) values(36, 'Nurmijärven Linja Oy');
insert into operators(id, name) values(40, 'HKL-Raitioliikenne');
insert into operators(id, name) values(45, 'Transdev Vantaa Oy');
insert into operators(id, name) values(47, 'Taksikuljetus Oy');
insert into operators(id, name) values(51, 'Korsisaari Oy');
insert into operators(id, name) values(54, 'V-S Bussipalvelut Oy');
insert into operators(id, name) values(55, 'Transdev Helsinki Oy');
insert into operators(id, name) values(58, 'Koillisen Liikennepalvelut Oy');
insert into operators(id, name) values(59, 'Tilausliikenne Nikkanen Oy');
insert into operators(id, name) values(90, 'VR Oy');
```

4. Double check the records are in the table:

```
postgres=# select * from operators;
 id |             name
----+------------------------------
  6 | Oy Pohjolan Liikenne Ab
 12 | Helsingin Bussiliikenne Oy
 17 | Tammelundin Liikenne Oy
 18 | Pohjolan Kaupunkiliikenne Oy
 19 | Etelä-Suomen Linjaliikenne Oy
 20 | Bus Travel Åbergin Linja Oy
 21 | Bus Travel Oy Reissu Ruoti
 22 | Nobina Finland Oy
 36 | Nurmijärven Linja Oy
 40 | HKL-Raitioliikenne
 45 | Transdev Vantaa Oy
 47 | Taksikuljetus Oy
 51 | Korsisaari Oy
 54 | V-S Bussipalvelut Oy
 55 | Transdev Helsinki Oy
 58 | Koillisen Liikennepalvelut Oy
 59 | Tilausliikenne Nikkanen Oy
 90 | VR Oy
(18 rows)
```


5. Exit `psql` and the container `postgres` by pressing `CTRL-d`.

If you prefer a graphical tool to edit your Postgres database then I recommend **PgAdmin4**. You can run this tool as a container as follows:

```
$ docker container run --rm -d \
    -p 8080:80 \
    --net using-connect_confluent \
    -e PGADMIN_DEFAULT_EMAIL=student@confluent.io \
    -e PGADMIN_DEFAULT_PASSWORD=TopSecret \
    dpage/pgadmin4
```

Then in your browser navigate to localhost:8080 and login with `student@confluent.io` and password `TopSecret`.
When defining a DB connection set **name** to **postgres**, **host** to **postgres**, **username** to **postgres** and leave the password field empty.

Configure the source connector

1. Create a new Topic called `pg-operators` with one partition and one replica:

```
$ kafka-topics \
    --bootstrap-server kafka:9092 \
    --create \
    --topic pg-operators \
    --partitions 1 \
    --replication-factor 1
```

2. Add a JDBC source connector via command line and **REST API** of Connect:

```
$ curl -s -X POST \
        -H "Content-Type: application/json" \
        --data '{
            "name": "Operators-Connector",
            "config": {
                "connector.class": "io.confluent.connect.jdbc.JdbcSourceConnector",
                "connection.url": "jdbc:postgresql://postgres:5432/postgres",
                "connection.user": "postgres",
                "table.whitelist": "operators",
                "mode":"incrementing",
                "incrementing.column.name": "id",
                "table.types": "TABLE",
                "topic.prefix": "pg-",
                "numeric.mapping": "best_fit",
                "transforms": "createKey,extractInt",
                "transforms.createKey.type":
"org.apache.kafka.connect.transforms.ValueToKey",
                "transforms.createKey.fields": "id",
                "transforms.extractInt.type":
"org.apache.kafka.connect.transforms.ExtractField$Key",
                "transforms.extractInt.field": "id"
            }
        }' http://connect:8083/connectors | jq
```

should give this output:


```
{
  "name": "Operators-Connector",
  "config": {
    "connector.class": "io.confluent.connect.jdbc.JdbcSourceConnector",
    "connection.url": "jdbc:postgresql://postgres:5432/postgres",
    "connection.user": "postgres",
    "table.whitelist": "operators",
    "mode": "incrementing",
    "incrementing.column.name": "id",
    "table.types": "TABLE",
    "topic.prefix": "pg-",
    "name": "JDBC-Source-Connector",
    "transforms": "createKey,extractInt",
    "transforms.createKey.type": "org.apache.kafka.connect.transforms.ValueToKey",
    "transforms.createKey.fields": "id",
    "transforms.extractInt.type": "org.apache.kafka.connect.transforms.ExtractField$Key",
    "transforms.extractInt.field": "id"
  },
  "tasks": [],
  "type": "source"
}
```

3. Let's see what we get:

```
$ kafka-avro-console-consumer \
      --bootstrap-server kafka:9092 \
      --property schema.registry.url=http://schema-registry:8081 \
      --topic pg-operators \
      --from-beginning \
      --property print.key=true
```

and we should see something like this:

```
6   {"id":6,"name":"Oy Pohjolan Liikenne Ab"}
12  {"id":12,"name":"Helsingin Bussiliikenne Oy"}
17  {"id":17,"name":"Tammelundin Liikenne Oy"}
18  {"id":18,"name":"Pohjolan Kaupunkiliikenne Oy"}
19  {"id":19,"name":"Etelä-Suomen Linjaliikenne Oy"}
20  {"id":20,"name":"Bus Travel Åbergin Linja Oy"}
21  {"id":21,"name":"Bus Travel Oy Reissu Ruoti"}
22  {"id":22,"name":"Nobina Finland Oy"}
36  {"id":36,"name":"Nurmijärven Linja Oy"}
40  {"id":40,"name":"HKL-Raitioliikenne"}
45  {"id":45,"name":"Transdev Vantaa Oy"}
47  {"id":47,"name":"Taksikuljetus Oy"}
51  {"id":51,"name":"Korsisaari Oy"}
54  {"id":54,"name":"V-S Bussipalvelut Oy"}
55  {"id":55,"name":"Transdev Helsinki Oy"}
58  {"id":58,"name":"Koillisen Liikennepalvelut Oy"}
59  {"id":59,"name":"Tilausliikenne Nikkanen Oy"}
90  {"id":90,"name":"VR Oy"}
```


Exit the consumer with `CTRL-C`.

Cleanup

1. Execute the following commands to completely cleanup your environment:

```
$ docker-compose down -v
```

Conclusion

In this exercise you have used a Kafka Connect JDBC source connector to import data residing in PostgreSQL database into the topic `pg_operators` in the Kafka cluster. This data can now e.g. be used to enrich our Vehicle Position data that we are getting from the **High-frequency Positioning** service of **digitransit**.

STOP HERE. THIS IS THE END OF THE EXERCISE.

Lab 07b Using the Syslog Connector

Using the Syslog Connector

The goal of this lab is to use the Kafka Connect Syslog source connector to import Syslog events into a topic in the Kafka cluster. This data can then be used to create monitoring pipelines.

Prerequisites

1. Navigate to the labs folder:

```
$ cd ~/confluent-dev/labs/using-syslog-connector
```

2. Build the custom Kafka Connect image containing the Syslog connector:

```
$ docker-compose build
```

This will download and install the Syslog connector from the Confluent hub (http://confluent.io/hub). You should see something like this:

```
zookeeper uses an image, skipping
kafka uses an image, skipping
schema-registry uses an image, skipping
Building connect
Step 1/3 : FROM confluentinc/cp-kafka-connect:5.3.0
 ---> a556d728ef1e
Step 2/3 : ENV CONNECT_PLUGIN_PATH="/usr/share/java,/usr/share/confluent-hub-components"
 ---> Using cache
 ---> 7b676ee465bc
Step 3/3 : RUN confluent-hub install --no-prompt confluentinc/kafka-connect-syslog:latest
 ---> Running in f94d3a2471b4
Running in a "--no-prompt" mode
Implicit acceptance of the license below:
Confluent Software Evaluation License
https://www.confluent.io/software-evaluation-license
Downloading component Kafka Connect Syslog 1.2.6, provided by Confluent, Inc. from
Confluent Hub and installing into /usr/share/confluent-hub-components
Adding installation directory to plugin path in the following files:
  /etc/kafka/connect-distributed.properties
  /etc/kafka/connect-standalone.properties
  /etc/schema-registry/connect-avro-distributed.properties
  /etc/schema-registry/connect-avro-standalone.properties

Completed
Removing intermediate container f94d3a2471b4
 ---> e7b4a9bf125e

Successfully built e7b4a9bf125e
Successfully tagged confluentinc/kafka-connect-mqtt:latest
```

3. **OPTIONAL:** Open the file `connect/Dockerfile` to see how we install the additional Syslog connector plugin into the Kafka Connect base image.
4. Run the Kafka cluster, including Confluent Control Center:

```
$ docker-compose up -d
```

Configuring and Using the Syslog Connector

1. Before we can use the Syslog connector we need to define the target topic on Kafka whose (default) name is `syslog`:

```
$ kafka-topics \
    --create \
    --bootstrap-server kafka:9092 \
    --partitions 6 \
    --replication-factor 1 \
    --topic syslog
```

2. Make sure the `connector` service is ready, e.g. by observing its log:

```
$ docker-compose logs -f connect
```

Press CTRL-C to stop following the log

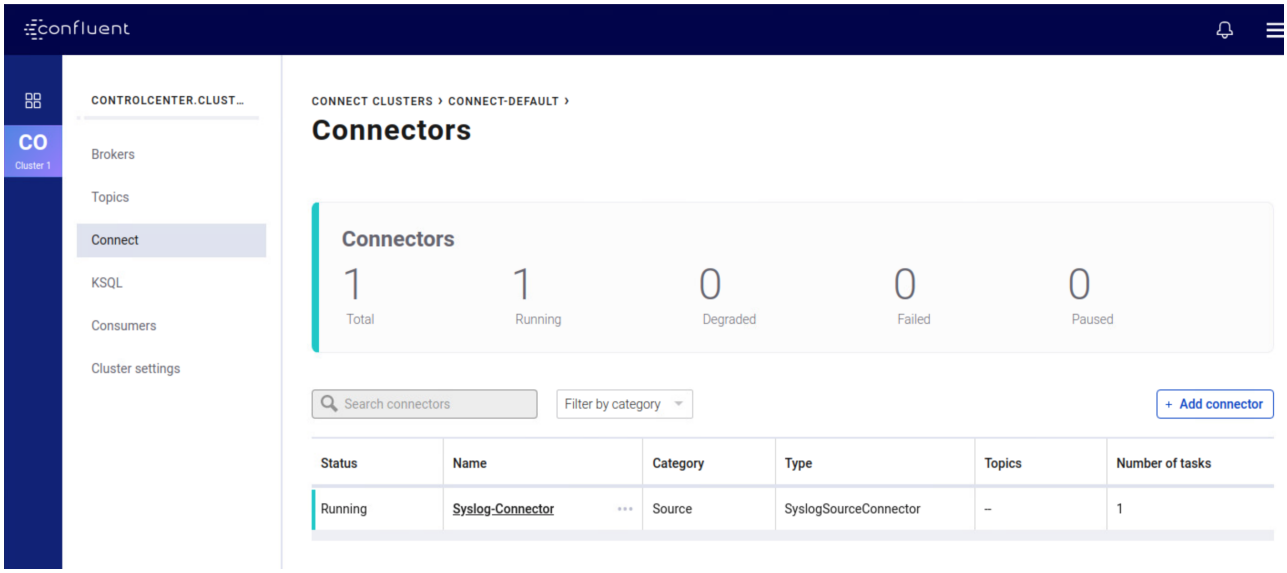
3. Once the `connect` service is ready, run the following command to configure the Syslog source connector in Kafka Connect:

```
$ curl -s -X POST \
        -H "Content-Type: application/json" \
        --data '{
            "name": "Syslog-Connector",
            "config": {
                "connector.class": "io.confluent.connect.syslog.SyslogSourceConnector",
                "tasks.max": "1",
                "syslog.port": "5454",
                "syslog.listener": "TCP",
                "confluent.topic.bootstrap.servers": "kafka:9092",
                "confluent.topic.replication.factor": "1",
                "confluent.license":""
            }
        }' http://connect:8083/connectors | jq
```


and you should see this output

```
{
  "name": "Syslog-Connector",
  "config": {
    "connector.class": "io.confluent.connect.syslog.SyslogSourceConnector",
    "tasks.max": "1",
    "syslog.port": "5454",
    "syslog.listener": "TCP",
    "confluent.license": "",
    "confluent.topic.bootstrap.servers": "kafka:9092",
    "confluent.topic.replication.factor": "1",
    "name": "Syslog-Connector"
  },
  "tasks": [],
  "type": "source"
}
```

4. In a new browser tab navigate to http://localhost:9021 to access Confluent Control Center:
   - Select cluster **Cluster 1**
   - then select the **Connect** tab. You will be shown the list of Connect clusters associated with your environment. There should only be a single one such cluster called `connect-default`
   - select the `connect-default` cluster and verify that there is indeed a source connector `Syslog-Connector` defined and that it is in status `running`:



5. Alternatively you can also list all connectors with this:

```
$ curl http://connect:8083/connectors
["Syslog-Connector"]
```

indicating that there is currently one connector configured (which of course was expected).

6. We can get the status of the `Syslog-Connector` connector and the associated task via:

```
$ curl -s "http://connect:8083/connectors/Syslog-Connector/status" | jq .
```

resulting in:

```
{
  "name": "Syslog-Connector",
  "connector": {
    "state": "RUNNING",
    "worker_id": "connect:8083"
  },
  "tasks": [
    {
      "id": 0,
      "state": "RUNNING",
      "worker_id": "connect:8083"
    }
  ],
  "type": "source"
}
```

7. Now let's test the connector by simulating a (syslog) message on port 5454 using the `netcat` (or `nc`) tool:
    - Output a test log entry to port 5454:

```
$ echo "<34>1 2003-10-11T22:14:15.003Z host1.acme.com su - ID47 - Engine started" \
    | nc -w 1 connect 5454
```

8. Let's verify that the entry has been written to the `syslog` topic:

```
$ kafka-avro-console-consumer \
    --bootstrap-server kafka:9092 \
    --property schema.registry.url=http://schema-registry:8081 \
    --topic syslog \
    --from-beginning | jq '.'
```

You should see this:

```
{
  "name": null,
  "type": "RFC5424",
  "message": {
    "string": "Engine started"
  },
  "host": {
    "string": "host1.acme.com"
  },
  "version": {
    "int": 1
  },
  "level": {
    "int": 2
  },
  "tag": null,
  "extension": null,
  "severity": null,
  "appName": {
    "string": "su"
  },
  "facility": {
    "int": 4
  },
  "remoteAddress": {
    "string": "127.0.0.1"
  },
  "rawMessage": {
    "string": "<34>1 2003-10-11T22:14:15.003Z host1.acme.com su - ID47 - Engine started"
  },
  "processId": null,
  "messageId": {
    "string": "ID47"
  },
  "structuredData": null,
  "deviceVendor": null,
  "deviceProduct": null,
  "deviceVersion": null,
  "deviceEventClassId": null,
  "date": 1065910455003
}
```


Press CTRL-C to exit the consumer.

Cleanup

1. Execute the following command to completely cleanup your environment:

```
$ docker-compose down -v
```

Conclusion

In this lab you have used the Kafka Connect Syslog source connector to import Syslog events into the topic `syslog` in the Kafka cluster. This data could be used to create monitoring and alerting pipeline.

STOP HERE. THIS IS THE END OF THE EXERCISE.

Lab 07c Using Kafka Connect with MQTT

Using Kafka Connect with MQTT

In this lab we want to use Kafka Connect to import data originating from our well known IoT data source. We will be using a MQTT source connector for this purpose. First we use **Mosquitto** to test the MQTT source.

For more details about the MQTT service see: https://digitransit.fi/en/developers/apis/4-realtime-api/vehicle-positions/

Testing the MQTT Data Source

1. Let's install **Mosquitto**, a MQTT client:

```
$ sudo apt-get update && \
    sudo apt-get install -y software-properties-common && \
    sudo apt-add-repository ppa:mosquitto-dev/mosquitto-ppa && \
    sudo apt-get install -y mosquitto-clients
```

2. To subscribe to the MQTT data provider we can use **Mosquitto's** subscription tool as follows:

```
$ mosquitto_sub -h mqtt.hsl.fi -p 1883 -d -t "/hfp/v2/journey/ongoing/vp/#"

Client mosq-8ZL8ZvKXQOPQsRkc7S sending CONNECT
Client mosq-8ZL8ZvKXQOPQsRkc7S received CONNACK (0)
Client mosq-8ZL8ZvKXQOPQsRkc7S sending SUBSCRIBE (Mid: 1, Topic:
/hfp/v2/journey/ongoing/vp/#, QoS: 0, Options: 0x00)
Client mosq-8ZL8ZvKXQOPQsRkc7S received SUBACK
Subscribed (mid: 1): 0
Client mosq-8ZL8ZvKXQOPQsRkc7S received PUBLISH (d0, q0, r0, m0,
'/hfp/v2/journey/ongoing/vp/bus/0045/01257/1055/1/Koskela/18:33/1020121/5/60;24/19/74/13'
, ... (303 bytes))
{"VP":{"desi":"55","dir":"1","oper":55,"veh":1257,"tst":"2019-10-
30T16:31:30.932Z","tsi":1572453090,"spd":0.00,"hdg":175,"lat":60.171618,"long":24.943079,
"acc":0.00,"dl":120,"odo":10,"drst":null,"oday":"2019-10-
30","jrn":679,"line":72,"start":"18:33","loc":"GPS","stop":1020121,"route":"1055","occu":
0}}
Client mosq-8ZL8ZvKXQOPQsRkc7S received PUBLISH (d0, q0, r0, m0,
'/hfp/v2/journey/ongoing/vp/bus/0012/01519/4560/2/Rastila
(M)/18:37/4150269/5/60;24/28/54/82', ... (300 bytes))
{"VP":{"desi":"560","dir":"2","oper":12,"veh":1519,"tst":"2019-10-
30T16:31:30.941Z","tsi":1572453090,"spd":0.00,"hdg":21,"lat":60.258551,"long":24.842830,"
acc":0.00,"dl":null,"odo":43,"drst":1,"oday":"2019-10-
30","jrn":1066,"line":743,"start":"18:37","loc":"GPS","stop":null,"route":"4560","occu":0
}}
...
```


As you can see, the tool connects successfully to the IoT data source and outputs records. We have shortened the output for readability. In the output we see the following:

- **Topic:** '/hfp/v2/journey/ongoing/vp/bus/0012/01519/4560/2/Rastila (M)/18:37/4150269/5/60;24/28/54/82'
- **Value:** {"VP":{"desi":"55","dir":"1","oper":55,"veh":1257,"tst":"2019-10-30T16:31:30.932Z","tsi":1572453090,"spd":0.00,"hdg":175,"lat":60.171618,"long":24.943079,"acc":0.00,"dl":120,"odo":10,"drst":null,"oday":"2019-10-30","jrn":679,"line":72,"start":"18:33","loc":"GPS","stop":1020121,"route":"1055","occu":0}}

3. When done, exit Mosquitto by pressing `CTRL-C`.

Using Kafka Connect

1. Navigate to the labs folder:

```
$ cd ~/confluent-dev/labs/mqtt-connect
```

2. Make sure the Docker image for Kafka Connect with the MQTT source connector is built:

```
$ docker-compose build
```

Open the Dockerfile at `connect/Dockerfile` and analyze how the Kafka Connect with the MQTT source connector image is defined.

3. Run the Kafka cluster, including Kafka Connect:

```
$ docker-compose up -d
```

4. Create a `vehicle-positions` topic on Kafka:

```
$ kafka-topics \
    --create \
    --bootstrap-server kafka:9092 \
    --partitions 6 \
    --replication-factor 1 \
    --topic vehicle-positions
```

5. Make sure Kafka Connect is up and running:

```
$ docker-compose logs -f connect | grep "INFO.*Finished starting connectors and tasks"
```

you should see something like this:

```
connect_1          | [2019-04-30 07:09:57,601] INFO [Worker clientId=connect-1,
groupId=connect] Finished starting ...
```

When you see the above output, press CTRL-C to stop following the log.

6. Create the Kafka Connect MQTT Source connector using the Connect REST API:

```
$ curl -s -X POST -H 'Content-Type: application/json' -d '{
    "name" : "mqtt-source",
    "config" : {
        "connector.class" : "io.confluent.connect.mqtt.MqttSourceConnector",
        "tasks.max" : "1",
        "mqtt.server.uri" : "tcp://mqtt.hsl.fi:1883",
        "mqtt.topics" : "/hfp/v2/journey/ongoing/vp/#",
        "kafka.topic" : "vehicle-positions",
        "confluent.topic.bootstrap.servers": "kafka:9092",
        "confluent.topic.replication.factor": "1",
        "confluent.license":"",
        "key.converter": "org.apache.kafka.connect.storage.StringConverter",
        "value.converter": "org.apache.kafka.connect.converters.ByteArrayConverter"
        }
    }' http://connect:8083/connectors | jq .
```

7. Check if connector is loaded and status is 'RUNNING':

```
$ curl -s "http://connect:8083/connectors"
["mqtt-source"]
```

and

```
$ curl -s "http://connect:8083/connectors/mqtt-source/status" | jq .
{
  "name": "mqtt-source",
  "connector": {
    "state": "RUNNING",
    "worker_id": "connect:8083"
  },
  "tasks": [
    {
      "id": 0,
      "state": "RUNNING",
      "worker_id": "connect:8083"
    }
  ],
  "type": "source"
}
```


Both, the state of the connector and the task should be RUNNING.

8. Let's see if some data is generated:

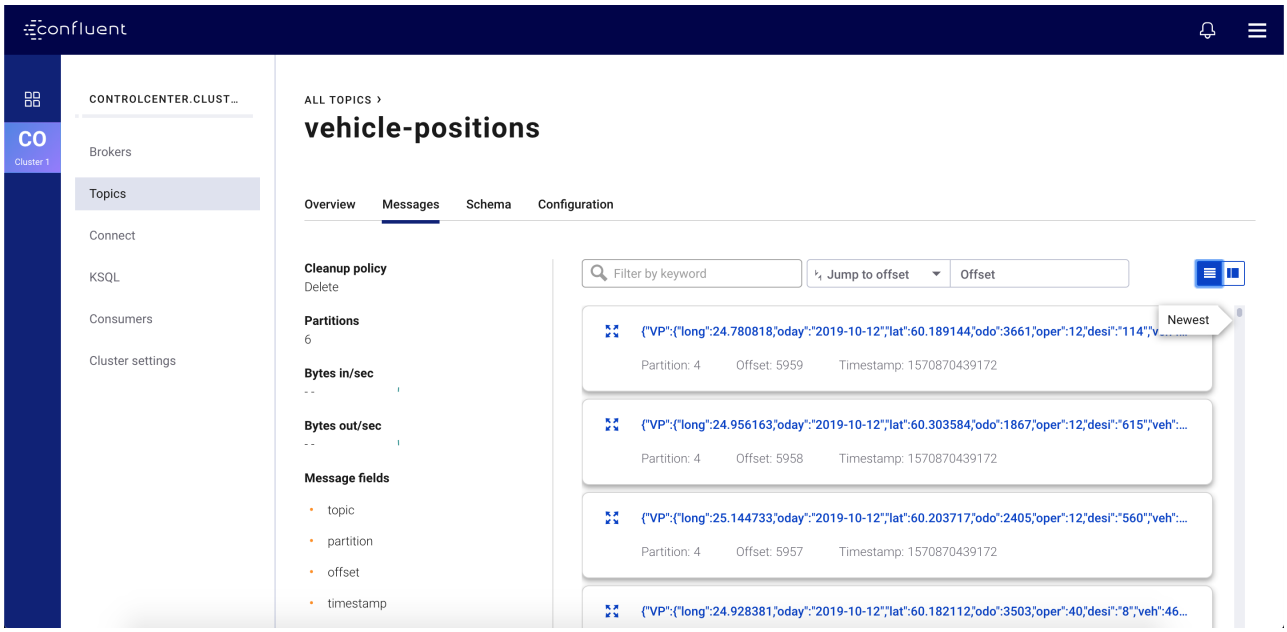
```
$ kafka-console-consumer --bootstrap-server kafka:9092 \
    --topic vehicle-positions \
    --from-beginning \
    --max-messages 5
```

and you should see an output similar to this:

```
{"VP":{"desi":"18","dir":"1","oper":12,"veh":1103,"tst":"2019-10-
30T18:18:57.756Z","tsi":1572459537,"spd":2.13,"hdg":95,"lat":60.203835,"long":24.869772,"
acc":0.66,"dl":77,"odo":null,"drst":null,"oday":"2019-10-
30","jrn":1005,"line":47,"start":"19:44","loc":"GPS","stop":1304144,"route":"1018","occu"
:0}}
{"VP":{"desi":"143","dir":"1","oper":18,"veh":436,"tst":"2019-10-
30T18:18:57.938Z","tsi":1572459537,"spd":0.13,"hdg":null,"lat":60.159774,"long":24.737697
,"acc":-0.01,"dl":300,"odo":null,"drst":0,"oday":"2019-10-
30","jrn":318,"line":231,"start":"20:23","loc":"GPS","stop":2314213,"route":"2143","occu"
:0}}
{"VP":{"desi":"4","dir":"1","oper":40,"veh":426,"tst":"2019-10-
30T18:18:57.989Z","tsi":1572459537,"spd":null,"hdg":null,"lat":null,"long":null,"acc":nul
l,"dl":282,"odo":186,"drst":0,"oday":"2019-10-
30","jrn":2396,"line":32,"start":"20:23","loc":"GPS","stop":1080416,"route":"1004","occu"
:0}}
{"VP":{"desi":"735","dir":"2","oper":22,"veh":630,"tst":"2019-10-
30T18:18:57.984Z","tsi":1572459537,"spd":0.00,"hdg":227,"lat":60.338567,"long":25.102546,
"acc":0.00,"dl":null,"odo":0,"drst":0,"oday":"2019-10-
30","jrn":708,"line":666,"start":"20:20","loc":"GPS","stop":4820213,"route":"4735","occu"
:0}}
{"VP":{"desi":"113","dir":"1","oper":12,"veh":1926,"tst":"2019-10-
30T18:18:58.038Z","tsi":1572459538,"spd":7.56,"hdg":272,"lat":60.217165,"long":24.825746,
"acc":-0.03,"dl":-55,"odo":7517,"drst":0,"oday":"2019-10-
30","jrn":455,"line":856,"start":"19:58","loc":"GPS","stop":2118284,"route":"2113","occu"
:0}}
Processed a total of 5 messages
```

demonstrating that indeed, the MQTT source connector did import vehicle positions from the source.

9. In a new browser tab navigate to http://localhost:9021 to access Confluent Control Center:
    - Select cluster **Cluster 1**
    - then select the **Topics** tab. You will see a list of topics defined on your cluster.
    - select the `vehicle-positions` topic and then in the details view select the **Messages** tab
    - switch to **card view** (on the right hand side) and expand one of the messages. You should see something like this:



Cleanup

1. Execute the following command to completely cleanup your environment:

```
$ docker-compose down -v
```


Conclusion

In this exercise you have used Kafka Connect with the MQTT source connector plugin to import IoT data into Kafka.

STOP HERE. THIS IS THE END OF THE EXERCISE.

Lab 07d OPTIONAL: Using the Confluent MQTT Proxy

Using the Confluent MQTT Proxy

In this (optional) lab we are going to use the Confluent MQTT Proxy to push IoT data to Kafka. At this time the MQTT proxy can only operate in push mode and not in subscriber mode.

For more details about the MQTT service see: https://digitransit.fi/en/developers/apis/4-realtime-api/vehicle-positions/

Prerequisites

1. If you haven't installed **Mosquitto** in the previous lab then follow these instructions:

```
$ sudo apt-get update && \
    sudo apt-get install -y software-properties-common && \
    sudo apt-add-repository ppa:mosquitto-dev/mosquitto-ppa && \
    sudo apt-get install -y mosquitto-clients
```

2. Navigate to the labs folder:

```
$ cd ~/confluent-dev/labs/mqtt-proxy
```

3. Run the Kafka cluster, including Kafka Connect:

```
$ docker-compose up -d
```


4. Open the file `docker-compose.yml` in your project folder and analyze the definition of the `mqtt` service:

```
29    mqtt:
30      image: confluentinc/cp-kafka-mqtt:5.3.0
31      hostname: mqtt
32      networks:
33        - confluent
34      ports:
35        - 1883:1883
36      environment:
37        KAFKA_MQTT_BOOTSTRAP_SERVERS: kafka:9092
38        KAFKA_MQTT_CONFLUENT_TOPIC_REPLICATION_FACTOR: 1
39        KAFKA_MQTT_TOPIC_REGEX_LIST: temperature:.*temperature, brightness:.*brightness
40        KAFKA_MQTT_LISTENERS: 0.0.0.0:1883
```

Specifically have an eye on how the environment variable KAFKA_MQTT_TOPIC_REGEX_LIST is defined, which maps MQTT topics to Kafka topics. In our case every MQTT topic that ends in `temperature` is mapped to the Kafka topic `temperature`. Every MQTT topic that ends in `brightness` is mapped to the Kafka topic

`brightness`.

Pushing IoT data to Kafka

1. Create a `temperatures` and a `brightness` topic on Kafka:

```
$ kafka-topics \
    --create \
    --bootstrap-server kafka:9092 \
    --partitions 1 \
    --replication-factor 1 \
    --topic temperature

$ kafka-topics \
    --create \
    --bootstrap-server kafka:9092 \
    --partitions 1 \
    --replication-factor 1 \
    --topic brightness
```

2. Use the Mosquitto client to publish some temperature records to the MQTT Proxy:

```
$ mosquitto_pub -h mqtt -p 1883 -t car/engine/temperature -q 2 -m "190F" && \
    mosquitto_pub -h mqtt -p 1883 -t car/engine/temperature -q 2 -m "200F" && \
    mosquitto_pub -h mqtt -p 1883 -t car/engine/temperature -q 2 -m "210F"
```

3. Now publish some brightness records:

```
$ mosquitto_pub -h mqtt -p 1883 -t car/lamp/brightness -q 2 -m "99lx" && \
    mosquitto_pub -h mqtt -p 1883 -t car/lamp/brightness -q 2 -m "98lx" && \
    mosquitto_pub -h mqtt -p 1883 -t car/lamp/brightness -q 2 -m "100lx" && \
    mosquitto_pub -h mqtt -p 1883 -t car/lamp/brightness -q 2 -m "98lx" && \
    mosquitto_pub -h mqtt -p 1883 -t car/lamp/brightness -q 2 -m "99lx"
```

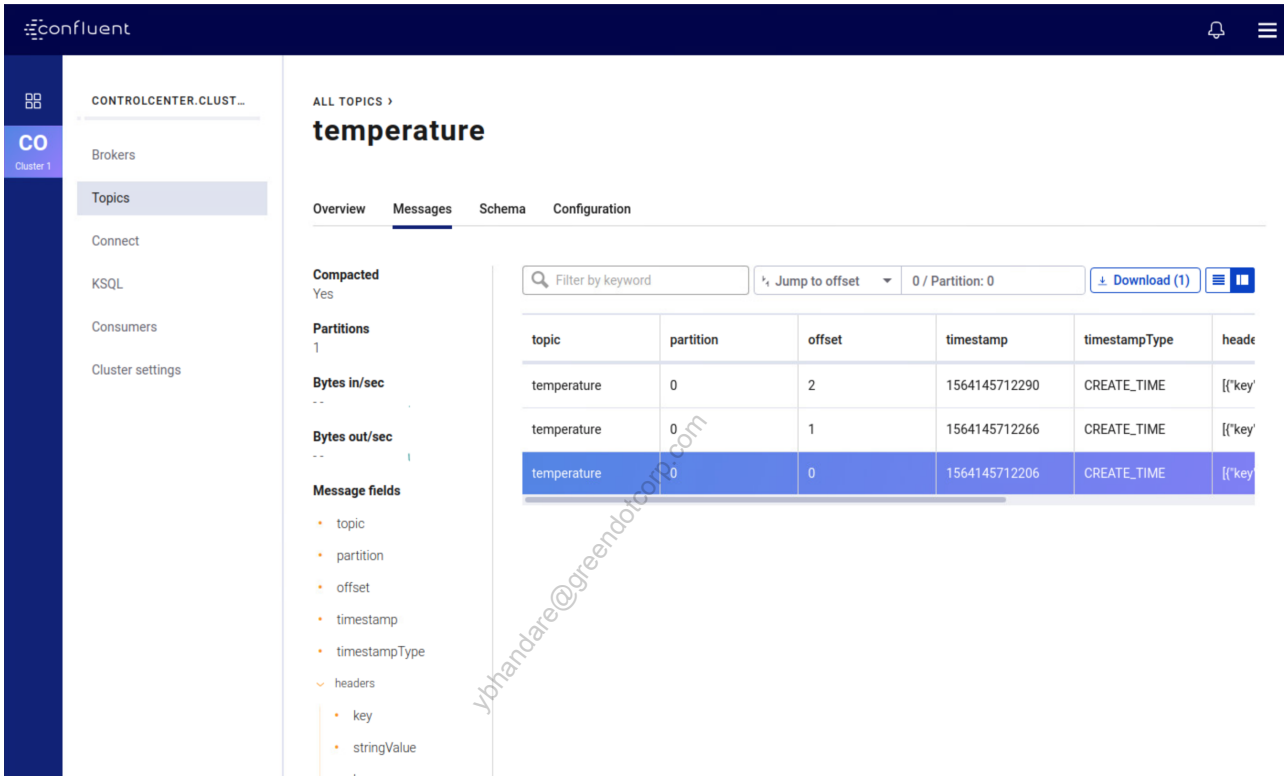
4. Use the Kafka console consumer to verify that the data has been written to Kafka:

```
$ kafka-console-consumer --bootstrap-server kafka:9092 \
    --topic temperature \
    --property print.key=true \
    --from-beginning
```

which should give you:

```
car/engine/temperature    190F
car/engine/temperature    200F
car/engine/temperature    210F
```

5. Do the same for the topic `brightness` and make sure you get the expected values.
6. In a new browser tab navigate to http://localhost:9021 to access Confluent Control Center:
    - Select cluster **Cluster 1**
    - then select the **Topics** tab. You will see a list of topics defined on your cluster.
    - select the `temperature` topic and then in the details view select the **Messages** tab
    - In the field **Jump to offset** enter `0` and hit enter. You should see something like this:



7. OPTIONAL: To generate an endless stream of readings use a script like this to generate temperatures:

```
$ while true; do echo $(( $RANDOM % (231-180) + 180)); sleep .2; done | \
    mosquitto_pub -h mqtt -p 1883 -t car/engine/temperature -q 2 -l
```

Hit CTRL-C to terminate the script, when done.

Cleanup

1. Execute the following command to completely cleanup your environment:

```
$ docker-compose down -v
```

Conclusion

In this exercise you have used the Confluent MQTT Proxy in push mode. Subscription mode is currently not supported but will come soon.

STOP HERE. THIS IS THE END OF THE EXERCISE.

Lab 08 Creating a Kafka Streams Application

Creating a Kafka Streams Application

In this exercise you are going to create a Kafka Streams Application in Java that will filter and transform data originating from the topic `vehicle-positions`. This topic in turn will be populated by a producer you have created in an earlier lab. The producer will consume the data from the **High-frequency Positioning** service of **digitransit**.

Prerequisites

1. Navigate to the project folder:

```
$ cd ~/confluent-dev/labs/streams-app
```

2. Run the Kafka cluster, including Confluent Control Center:

```
$ docker-compose up -d
```

3. Create the topic `vehicle-positions` with 6 partitions on Kafka with this command:

```
$ kafka-topics \
    --create \
    --bootstrap-server kafka:9092 \
    --partitions 6 \
    --replication-factor 1 \
    --topic vehicle-positions
```


4. Create the topic `vehicle-positions-oper-47` with 6 partitions on Kafka:

```
$ kafka-topics \
    --create \
    --bootstrap-server kafka:9092 \
    --partitions 6 \
    --replication-factor 1 \
    --topic vehicle-positions-oper-47
```

   This is the output topic for the Kafka Streams application we're going to build. It will contain only vehicle positions of operator 47.

5. Run a Docker container with the Kafka producer from the lab **Creating a Kafka Producer**:

```
$ docker container run --rm -d \
    --name vp-producer \
    --hostname vp-producer \
    --net labs_confluent \
    cnfltraining/vp-producer:v2
```

If you were not able to finish the lab **Creating a Kafka Producer** then don't worry. The necessary image `cnfltraining/vp-producer:v2` will just be downloaded from our Confluent Training org on Docker Hub.

If you encounter problems with the producer, e.g. the MQTT datasource that the producer is relying on is down, then please use the Docker image `cnfltraining/vp-producer-fallback:v2` instead.

6. Open the project in Visual Studio Code:

```
$ code .
```

Writing the Kafka Streams Application

1. Please note the three Java classes in folder `src/main/java/clients` called `VehiclePositionTransformer.java`, `VehicleValue.java` and `VehiclePosition.java`. The latter is a POJO used to deserialize the vehicle position record from JSON.
2. Let's now define the Kafka Streams application by implementing the `VehiclePositionTransformer.java` class. In the `main` method define the configuration for the app as follows:

```
Properties settings = new Properties();
settings.put(StreamsConfig.APPLICATION_ID_CONFIG, "vp-streams-app");
settings.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, "kafka:9092");
```

   This is the minimum needed to configure a Kafka Streams application. We need to provide an **application ID** and a list of URIs to the **Kafka bootstrap servers**. Since we have only a single broker, we provide its URI `kafka:9092`.

3. Next we get the topology for our Kafka Streams application and use this topology together with the configuration object to create a `KafkaStreams` object:

```
Topology topology = getTopology();
KafkaStreams streams = new KafkaStreams(topology, settings);
```

   We will implement the `getTopology` method in a subsequent step.

4. Now we define the shutdown behavior of the application:

```
Runtime.getRuntime().addShutdownHook(new Thread(() -> {
    System.out.println("<<< Stopping the vp-streams-app Application");
    streams.close();
}));
```

Here the important thing is to `close` the `streams` object.

5. Finally we can start the streams application:

```
streams.start();
```

And that's all for the `main` method.

6. Next we address the `getTopology` method. First we define the SerDes we're going to use for the `key` and `value` part of the Kafka records:

```
final Serde<String> stringSerde = Serdes.String();
final Serde<VehiclePosition> vpSerde = getJsonSerde();
```

We will use the `stringSerde` for the `key` and the `vpSerde` for the `value`.

7. Next we need a builder object helping us to create the topology:

```
StreamsBuilder builder = new StreamsBuilder();
```

8. We can use the builder's `stream` method to consume messages from our source topic `vehicle-positions`:

```
KStream<String,VehiclePosition> positions = builder
    .stream("vehicle-positions", Consumed.with(stringSerde, vpSerde));
```

The topic we consume from is `vehicle-positions`, and the SerDes for **key** and **value** are provided via `Consumed.with(...)` to the `stream` method.

9. Now let's filter the source stream and only let records through whose operator is equal to `47`:

```
KStream<String,VehiclePosition> operator47Only =
    positions.filter((key,value) -> value.VP.oper == 47);
```

10. Finally we want to output the filtered stream to the topic `vehicle-positions-oper-47`:

```
operator47Only.to("vehicle-positions-oper-47",
                  Produced.with(stringSerde, vpSerde));
```

We output the `value` also as JSON and thus can use the `vpSerde` object.

11. As a last step we need to build the stream processing topology and return it from the method:

```
Topology topology = builder.build();
return topology;
```

12. The last method to implement is the `getJsonSerde`. It's body looks like this:

```
Map<String, Object> serdeProps = new HashMap<>();
serdeProps.put("json.value.type", VehiclePosition.class);
final Serializer<VehiclePosition> vpSerializer = new KafkaJsonSerializer<>();
vpSerializer.configure(serdeProps, false);

final Deserializer<VehiclePosition> vpDeserializer = new KafkaJsonDeserializer<>();
vpDeserializer.configure(serdeProps, false);
return Serdes.serdeFrom(vpSerializer, vpDeserializer);
```

The code is somewhat self explaining. We're creating a serializer and deserializer object. They are both using the standard Kafka Json (De-)Serializer. The class to serialize from or deserialize to is of course our `VehiclePosition` class. With this serializer/deserializer pair we then create a SerDes and return it.

13. Now, in VS Code click the menu **Debug → Start Debugging** to start debugging. Observe the following output in the **DEBUG CONSOLE**:

```
>>> Starting the vp-streams-app Application
```

14. Run the `kafka-console-consumer` tool to verify that we have data in the target topic, and that you only see entries which have a value of 47 in the `oper` field:

```
$ kafka-console-consumer \
    --bootstrap-server kafka:9092 \
    --topic vehicle-positions-oper-47 \
    --from-beginning
```

which should result in something similar to this:

```
...
{"VP":{"desi":"544","dir":"2","oper":47,"veh":17,"tst":"2019-10-
30T04:10:16.215Z","tsi":1572408616,"spd":20.72,"hdg":91,"lat":60.20598,"acc":-1.89,"dl":-
3,"odo":14823,"drst":0,"oday":"2019-10-
30","jrn":54,"line":1039,"start":"05:45","loc":"GPS","stop":null,"route":"2544","occu":0,
"seq":0,"long":24.764395}}
{"VP":{"desi":"818","dir":"1","oper":47,"veh":801,"tst":"2019-10-
30T04:10:13.340Z","tsi":1572408613,"spd":4.93,"hdg":344,"lat":60.213028,"acc":0.15,"dl":-
60,"odo":3455,"drst":0,"oday":"2019-10-
30","jrn":3,"line":598,"start":"06:03","loc":"GPS","stop":null,"route":"1818","occu":0,"s
eq":0,"long":25.175093}}
{"VP":{"desi":"813","dir":"2","oper":47,"veh":814,"tst":"2019-10-
30T04:10:16.226Z","tsi":1572408616,"spd":2.45,"hdg":265,"lat":60.221928,"acc":1.23,"dl":0
,"odo":556,"drst":0,"oday":"2019-10-
30","jrn":151,"line":593,"start":"06:09","loc":"GPS","stop":"1541047","route":"1813","occ
u":0,"seq":0,"long":25.13389}}
...
```

If you don't get any data, it might be that operator 47 has no lines in operation at that time. In this case you may retry with a different operator number.

15. Alternatively, in your browser navigate to http://localhost:9021 to access Confluent Control Center:
    a. select cluster **Cluster 1**
    b. select the tab **Topics**
    c. from the list of topics select `vehicle-positions-oper-47`
    d. switch to the **Messages** tab
    e. verify that you only see entries which have a value of 47 in the `oper` field.
16. In VS Code stop the debugger.
17. Stop the producer with:

```
$ docker kill vp-producer
```

OPTIONAL: Building a Docker Image for the Kafka Streams Application

To prepare for the next lab we want to create a Docker image containing our Kafka Stream application, so we can easily run it as a part of our distributed application.

1. In VS Code open the file `Dockerfile` in the project folder and observe the steps needed to containerize our Java application. Specifically note that, in order to reduce the size of the final image, we are using a multi-step build process.
2. In the terminal, from within the project folder execute the following command to build the image:

```
$ docker image build -t cnfltraining/vp-streams-app:v2 .
```

don't forget the period (`.`) at the end of the above command! It instructs the Docker builder to use the current directory as the context.

3. Test the new image by running a container from it:

```
$ docker container run --rm -it \
    --net labs_confluent \
    cnfltraining/vp-streams-app:v2
```

You should see an output like this:

```
>>> Starting Sample Streams Application
```

4. Stop the Kafka Streams app by pressing `CTRL-C`.

The complete solution can be found at `~/confluent-dev/solution/streams-app`.

Cleanup

1. Execute the following command to completely cleanup your environment:

```
$ docker-compose down -v
```

Conclusion

In this exercise you have created a Kafka Streams Application in Java that filters data originating from the topic `vehicle-positions`. This topic is populated by a producer you have created in an earlier lab. The producer consumes the data from the **High-frequency Positioning** service of **digitransit**.

STOP HERE. THIS IS THE END OF THE EXERCISE.

Lab 09 Using Confluent KSQL

Using Confluent KSQL

The goal of this lab is to filter, transform and aggregate data using KSQL. This data is created by our simple Kafka producer, that we created earlier. It stems from live data of the **High-frequency Positioning** service of **digitransit**.

Prerequisites

1. Navigate to the project folder:

```
$ cd ~/confluent-dev/labs/using-ksql
```

2. Run the Kafka cluster, including Confluent Control Center:

```
$ docker-compose -p labs up -d
```

3. Make sure the topic `vehicle-positions` has been created by following the log of service `base`:

```
$ docker-compose -p labs logs -f base
```


Eventually you should see the output:

```
...
Creating the topics...
base_1             | Created topic "vehicle-positions".
base_1             | Created topic "operators".
base_1             | Created topic operators.
base_1             | >>>>>>>>>>>>>>>>>>>>labs_base_1 exited with code 0
```

4. Verify that there is data in the topic `operators`:

```
$ kafka-console-consumer --bootstrap-server kafka:9092 \
    --topic operators \
    --from-beginning \
    --property print.key=true
```

And you should see:

```
6       {"id": 6, "name": "Oy Pohjolan Liikenne Ab"}
12      {"id": 12, "name": "Helsingin Bussiliikenne Oy"}
17      {"id": 17, "name": "Tammelundin Liikenne Oy"}
18      {"id": 18, "name": "Pohjolan Kaupunkiliikenne Oy"}
19      {"id": 19, "name": "Etelä-Suomen Linjaliikenne Oy"}
20      {"id": 20, "name": "Bus Travel Åbergin Linja Oy"}
21      {"id": 21, "name": "Bus Travel Oy Reissu Ruoti"}
22      {"id": 22, "name": "Nobina Finland Oy"}
36      {"id": 36, "name": "Nurmijärven Linja Oy"}
40      {"id": 40, "name": "HKL-Raitioliikenne"}
45      {"id": 45, "name": "Transdev Vantaa Oy"}
47      {"id": 47, "name": "Taksikuljetus Oy"}
51      {"id": 51, "name": "Korsisaari Oy"}
54      {"id": 54, "name": "V-S Bussipalvelut Oy"}
55      {"id": 55, "name": "Transdev Helsinki Oy"}
58      {"id": 58, "name": "Koillisen Liikennepalvelut Oy"}
59      {"id": 59, "name": "Tilausliikenne Nikkanen Oy"}
90      {"id": 90, "name": "VR Oy"}
```

Exit the consumer with CTRL-C

5. Run the Kafka clients (producer and consumer) that you had created in previous labs:

```
$ docker-compose -p labs -f docker-compose-clients.yml up -d

WARNING: Found orphan containers (using-ksql_kafka_1, using-ksql_control-center_1, using-
ksql_schema-registry_1, using-ksql_zookeeper_1) for this project. If you removed or
renamed this service in your compose file, you can run this command with the --remove
-orphans flag to clean it up.
using-ksql_vp-consumer_1 is up-to-date
using-ksql_vp-producer_1 is up-to-date
```


If you did not complete the previous labs where we created the producer and the consumer then don't worry. When running the file `docker-compose-clients.yml` the images will simply be downloaded from Docker Hub.

By the way, you can safely ignore the warning in the output above.

Using KSQL to analyze Data

1. Execute into the KSQL CLI:

```
$ ksql http://ksql-server:8088
...
Copyright 2017-2018 Confluent Inc.

CLI v5.3.0, Server v5.3.0 located at http://ksql-server:8088

Having trouble? Type 'help' (case-insensitive) for a rundown of how things work!

ksql>
```

2. Now create a stream from the topic `vehicle-positions`:

```
ksql> CREATE STREAM vehicle_positions(
    VP STRUCT<
        desi  STRING,
        dir  STRING,
        oper  INTEGER,
        veh  INTEGER,
        tst  STRING,
        tsi  BIGINT,
        spd  DOUBLE,
        hdg  INTEGER,
        lat  DOUBLE,
        long  DOUBLE,
        acc  DOUBLE,
        dl  INTEGER,
        odo  INTEGER,
        drst  INTEGER,
        oday  STRING,
        jrn  INTEGER,
        line  INTEGER,
        start  STRING,
        loc  STRING,
        stop  STRING,
        route  STRING,
        occu  INTEGER,
        seq  INTEGER
    >
    ) WITH(KAFKA_TOPIC='vehicle-positions', VALUE_FORMAT='JSON');
```


Note how we're using a STRUCT to work with nested JSON data.

3. Try to run a query against this stream:

```
ksql> SELECT VP->oper, VP->veh, VP->tst, VP->lat, VP->long
    FROM vehicle_positions
    LIMIT 10;
```

Again note how we use e.g. VP¬`oper` to access the `oper` field of the struct.

In my case the result looks like this:

```
22 | 883 | 2019-03-13T14:23:28Z | 60.235853 | 25.082712
40 | 453 | 2019-03-13T14:23:28Z | 60.199328 | 24.93419
12 | 1401 | 2019-03-13T14:23:28Z | 60.205172 | 24.899306
18 | 761 | 2019-03-13T14:23:28Z | 60.183712 | 24.831706
90 | 6059 | 2019-03-13T14:23:28Z | 60.481608 | 25.076404
18 | 672 | 2019-03-13T14:23:28Z | 60.220789 | 24.932016
12 | 905 | 2019-03-13T14:23:28Z | 60.205014 | 24.880521
12 | 1505 | 2019-03-13T14:23:28Z | 60.262981 | 24.871294
18 | 658 | 2019-03-13T14:23:28Z | 60.188923 | 24.963767
18 | 266 | 2019-03-13T14:23:26Z | null | null
Limit Reached
Query terminated
```

If you don't get any data, then this could have different reasons, some of them are:

- There is no incoming data in the topic at this time
- You did not specify that the query should start from `earliest` (see later)
- Event/records may be skipped due to deserialization errors

In the latter case verify your assumption with the following query:

```
$ docker-compose -p labs exec ksql-server \
    cat ./usr/logs/ksql-streams.log \
    | grep "deserialization error"
```


4. To apply filtering you can use the `WHERE` statement. Let's just output records from operator `90`:

```
ksql> SELECT VP->oper, VP->veh, VP->tst, VP->lat, VP->long
    FROM vehicle_positions
    WHERE VP->oper = 90
    LIMIT 10;
90 | 6073 | 2019-03-13T14:25:54Z | 60.787528 | 25.48345
90 | 1043 | 2019-03-13T14:25:54Z | 60.405136 | 25.106801
90 | 1072 | 2019-03-13T14:25:54Z | 60.172844 | 24.939983
90 | 1001 | 2019-03-13T14:25:54Z | 60.174178 | 24.939463
90 | 6052 | 2019-03-13T14:25:54Z | 60.171702 | 24.94185
90 | 1080 | 2019-03-13T14:25:54Z | 60.138411 | 24.261195
90 | 6324 | 2019-03-13T14:25:54Z | 60.172063 | 24.941604
90 | 6059 | 2019-03-13T14:25:54Z | 60.489454 | 25.062152
90 | 1016 | 2019-03-13T14:25:54Z | 60.174535 | 24.94179
90 | 1073 | 2019-03-13T14:25:54Z | null | null
Limit Reached
Query terminated
```

If the above query does not output any data then think about why. What time is it right now in Finland?
Try to use a filter by a different operator (`oper`) in that case.

5. We can also use multiple filter criteria combined with AND or OR. Further filter for vehicle numbers smaller than 2000:

```
ksql> SELECT VP->oper, VP->veh, VP->tst, VP->lat, VP->long
    FROM vehicle_positions
    WHERE VP->oper = 90 AND VP->veh < 2000
    LIMIT 10;
90 | 1066 | 2019-03-13T14:28:07Z | 60.209306 | 24.917688
90 | 1079 | 2019-03-13T14:28:07Z | 60.301972 | 25.048682
90 | 1007 | 2019-03-13T14:28:07Z | 60.351763 | 25.078787
90 | 1038 | 2019-03-13T14:28:07Z | 60.246443 | 24.864041
90 | 1043 | 2019-03-13T14:28:07Z | 60.405137 | 25.106802
90 | 1070 | 2019-03-13T14:28:07Z | 60.219162 | 24.813751
90 | 1050 | 2019-03-13T14:28:07Z | 60.177369 | 24.93972
90 | 1063 | 2019-03-13T14:28:07Z | 60.219089 | 24.815167
90 | 1056 | 2019-03-13T14:28:07Z | 60.286422 | 25.041523
90 | 1023 | 2019-03-13T14:28:07Z | 60.173441 | 24.940477
```

Aggregating and Windowing Data

1. Now let's see the max vehicle speed per vehicle per minute, for operator 90, vehicle number in the range [1000..1100] and vehicle speed must be defined:


```
ksql> SELECT WINDOWSTART(), WINDOWEND(),
             VP->veh, MAX(VP->spd) AS max_spd
    FROM vehicle_positions
    WINDOW TUMBLING (SIZE 1 MINUTE)
    WHERE VP->oper = 90
        AND VP->veh >= 1000 AND VP->veh < 1100
        AND VP->spd IS NOT NULL
    GROUP BY VP->veh;
```

You should see something like this:

```
...
1561987740000 | 1561987800000 | 1072 | 0.33
1561987740000 | 1561987800000 | 1005 | 3.23
1561987740000 | 1561987800000 | 1063 | 16.78
1561987740000 | 1561987800000 | 1019 | 17.19
1561987740000 | 1561987800000 | 1048 | 0.0
1561987740000 | 1561987800000 | 1037 | 17.93
1561987740000 | 1561987800000 | 1001 | 13.34
1561987740000 | 1561987800000 | 1015 | 15.34
...
```

2. Stop the query with CTRL-C.

Joining Tables and Streams

1. Create a table `operators` from the topic `operators` as follows:

```
ksql> CREATE TABLE operators(
    id STRING,
    name STRING
) WITH(KAFKA_TOPIC='operators', VALUE_FORMAT='JSON', KEY='id');
```

2. Set the property `auto.offset.reset` to `earliest` such as that KSQL returns data from the very beginning of a table or stream when querying:

```
ksql> set 'auto.offset.reset'='earliest';
```

3. Verify that there is data in the table `operators`:

```
ksql> SELECT * FROM operators;

1558956376883 | 6 | 6 | Oy Pohjolan Liikenne Ab
1558956376906 | 12 | 12 | Helsingin Bussiliikenne Oy
1558956376906 | 17 | 17 | Tammelundin Liikenne Oy
1558956376906 | 18 | 18 | Pohjolan Kaupunkiliikenne Oy
1558956376907 | 19 | 19 | Etelä-Suomen Linjaliikenne Oy
1558956376907 | 20 | 20 | Bus Travel Åbergin Linja Oy
1558956376907 | 21 | 21 | Bus Travel Oy Reissu Ruoti
1558956376907 | 22 | 22 | Nobina Finland Oy
1558956376908 | 36 | 36 | Nurmijärven Linja Oy
1558956376908 | 40 | 40 | HKL-Raitioliikenne
1558956376909 | 45 | 45 | Transdev Vantaa Oy
1558956376909 | 47 | 47 | Taksikuljetus Oy
1558956376909 | 51 | 51 | Korsisaari Oy
1558956376909 | 54 | 54 | V-S Bussipalvelut Oy
1558956376910 | 55 | 55 | Transdev Helsinki Oy
1558956376910 | 58 | 58 | Koillisen Liikennepalvelut Oy
1558956376910 | 59 | 59 | Tilausliikenne Nikkanen Oy
1558956376910 | 90 | 90 | VR Oy
```

   and stop the query with `CTRL-C`.

To enrich data we can join streams to streams or tables.

1. Before we can join the table `operators` with the stream `vehicle_positions` we need to repartition the stream `vehicle_positions` and co-partition it with the `operators` table. We can do that as follows:

```
ksql> CREATE STREAM positions WITH (PARTITIONS=1) AS
      SELECT VP->oper AS oper,
             VP->veh AS veh,
             VP->desi AS desi,
             VP->dir AS dir,
             VP->spd AS spd
      FROM vehicle_positions
      PARTITION BY oper;
```

Co-partitioning means that the stream and the table need to have the same key and the same amount of partitions (1 in our case).

It is also not possible to join on a nested struct field. So not only do we need to repartition the data to satisfy co-partitioning requirements, we need to flatten the data before we can use the operator (`oper`) field in a join.

2. Now let's do the join between the stream and the table:

```
ksql> SELECT o.name, p.veh, p.desi, p.dir, p.spd
    FROM positions p
    JOIN operators o
    ON p.oper=o.id
    LIMIT 10;

Helsingin Bussiliikenne Oy | 1809 | 621 | 2 | 0.28
Nobina Finland Oy | 863 | 243 | 2 | 11.89
VR Oy | 1017 | I | 1 | 0.0
Pohjolan Kaupunkiliikenne Oy | 627 | 56 | 2 | 1.16
Nobina Finland Oy | 828 | 203 | 1 | 14.55
Helsingin Bussiliikenne Oy | 1524 | 39B | 1 | 0.0
Nobina Finland Oy | 820 | 731 | 2 | 12.09
Helsingin Bussiliikenne Oy | 821 | 42 | 2 | 8.73
Nobina Finland Oy | 985 | 624 | 2 | 5.0
Nobina Finland Oy | 945 | 731 | 1 | 10.87
Limit Reached
Query terminated
```


Cleanup

1. Exit the KSQL CLI by pressing CTRL-d.
2. Execute the following commands to completely cleanup your environment:

```
$ docker-compose -p labs -f docker-compose-clients.yml down
$ docker-compose -p labs down -v
```

Conclusion

In this exercise you have used KSQL to define streams and tables that you could then transform, filter and aggregate. You have also used KSQL to join a stream with a table. To do that you first had to co-partition the stream with the table.

STOP HERE. THIS IS THE END OF THE EXERCISE.

Lab 10 Writing a Microservice

Writing a Microservice

In this lab we're going to write a microservice that will consume events from a Kafka topic and use them to trigger certain actions. Those actions in turn will raise new events that are published back to Kafka into a different topic.

Prerequisites

1. Start the Kafka cluster

```
$ cd ~/confluent-dev/labs/microservice
$ docker-compose up -d
```

2. Create the topic `vehicle-positions`

```
$ kafka-topics \
    --create \
    --bootstrap-server kafka:9092 \
    --partitions 6 \
    --replication-factor 1 \
    --topic vehicle-positions
```

If the topic already exists then you get an error:
`Error while executing topic command : Topic vehicle-positions already exists....`

3. Run a Docker container with the Kafka producer from the lab **Creating a Kafka Producer**:

```
$ docker container run --rm -d \
    --name producer \
    --net labs_confluent \
    cnfltraining/vp-producer:v2
```

If you were not able to finish the lab **Creating a Kafka Producer** then don't worry. The necessary image `cnfltraining/vp-producer:v2` will just be downloaded from our Confluent Training org on Docker Hub.

If you encounter problems with the producer, e.g. the MQTT datasource that the producer is relying on is down, then please use the Docker image `cnfltraining/vp-producer-fallback:v2` instead.

Creating a persistent Query

1. Start KSQL:

```
$ ksql http://ksql-server:8088
```

2. In KSQL create a stream called `vehicle_positions` from the topic `vehicle-positions`

```
ksql> CREATE STREAM vehicle_positions(
    VP STRUCT<
        desi  STRING,
        dir  STRING,
        oper  INTEGER,
        veh  INTEGER,
        tst  STRING,
        tsi  BIGINT,
        spd  DOUBLE,
        hdg  INTEGER,
        lat  DOUBLE,
        long  DOUBLE,
        acc  DOUBLE,
        dl  INTEGER,
        odo  INTEGER,
        drst  INTEGER,
        oday  STRING,
        jrn  INTEGER,
        line  INTEGER,
        start  STRING,
        loc  STRING,
        stop  STRING,
        route  STRING,
        occu  INTEGER,
        seq  INTEGER
    >
    ) WITH(KAFKA_TOPIC='vehicle-positions', VALUE_FORMAT='JSON');
```


3. Create a stream `tram_door_status`:

```
ksql> CREATE STREAM tram_door_status AS
    SELECT vp->oper AS oper,
           vp->desi AS desi,
           vp->veh AS veh,
           vp->drst AS drst
    FROM vehicle_positions
    WHERE ROWKEY LIKE '%/tram/%';
```

Note how the new stream only contains records of trams and only the fields

- operator (oper)

- route number (desi)
- Vehicle number (veh)
- Door status (drst)

If the query does not return any data, it may be due to the fact that no trams are running at this time (what is the current time in Finland right now?). In this case you may try another filter, and maybe look for **trains**.

Creating the Microservice

The goal is to write a microservice in Java that consumes the topic `TRAM_DOOR_STATUS` produced by the persisted KSQL query you defined above. The service should discover for each vehicle when its door status changes. It should trigger an event `DoorsOpened` when the door status of a vehicle changes from 0 to 1, and it should trigger an event `DoorsClosed` when the opposite happens. The microservice writes the events into a topic `tram-door-status-changed`.

The event should be mapped to a Kafka record as follows:

record value:

```
{
    "operator": "<operator>",
    "route": "<route number>",
    "vehicle": "<vehicle>",
    "type": "DOOR_OPENED|DOOR_CLOSED"
}
```


1. Create the topic `tram-door-status-changed`:

```
$ kafka-topics \
    --create \
    --bootstrap-server kafka:9092 \
    --partitions 1 \
    --replication-factor 1 \
    --topic tram-door-status-changed
```

2. Run the microservice
3. In another terminal window start the `kafka-console-consumer` for the topic `tram-door-status-changed` to monitor the incoming events
4. Alternatively, observe the inflowing messages in Confluent Control Center, which you can access in a new browser tab on http://localhost:9021. Navigate to the topic `tram-door-status-changed` and then select the **Messages** tab.

If you find it difficult to come up with your own solution then first try to discuss a solution with your peers, and second have a look into the sample solution at `~/confluent-dev/solution/microservice`.

Cleanup

1. Exit KSQL by pressing CTRL-d
2. Stop your microservice
3. Execute the following commands to tear down the producer and the Kafka cluster:

```
$ docker container rm -f producer
$ docker-compose down -v
```

Conclusion

In this exercise you have authored a microservice that reacts on events delivered by Kafka and publishes events back to Kafka.

STOP HERE. THIS IS THE END OF THE EXERCISE.

Lab 11 OPTIONAL: Using Confluent Cloud

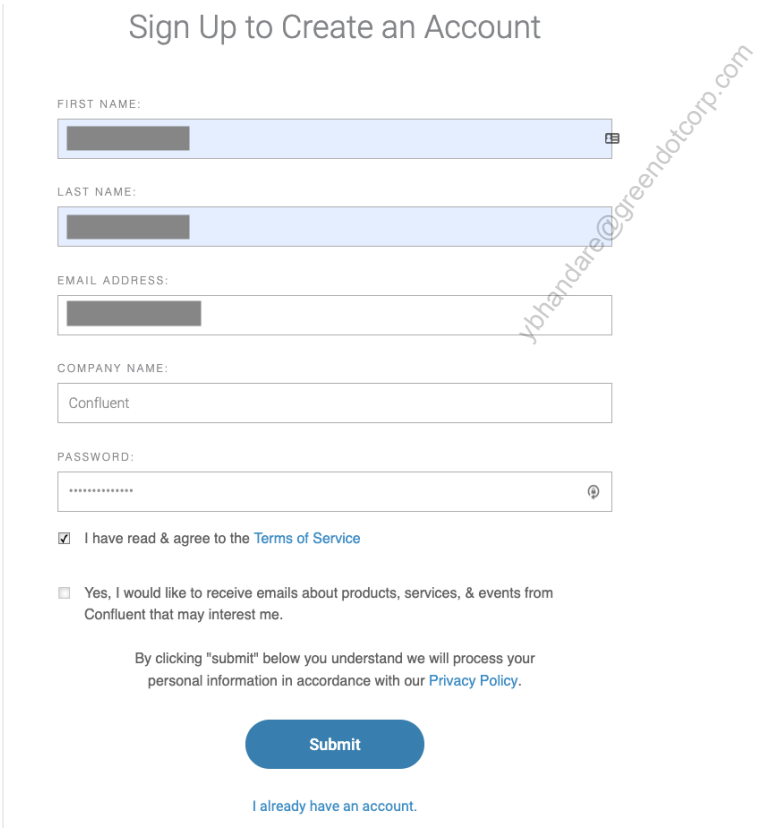
Using Confluent Cloud

In this lab we are going to create an account on Confluent Cloud (CC) and will run a simple hosted Kafka cluster on it.

Currently Confluent Cloud only offers paid access to hosted Kafka. You need a Credit Card to create an account on Confluent Cloud. This is the reason we make this lab optional.

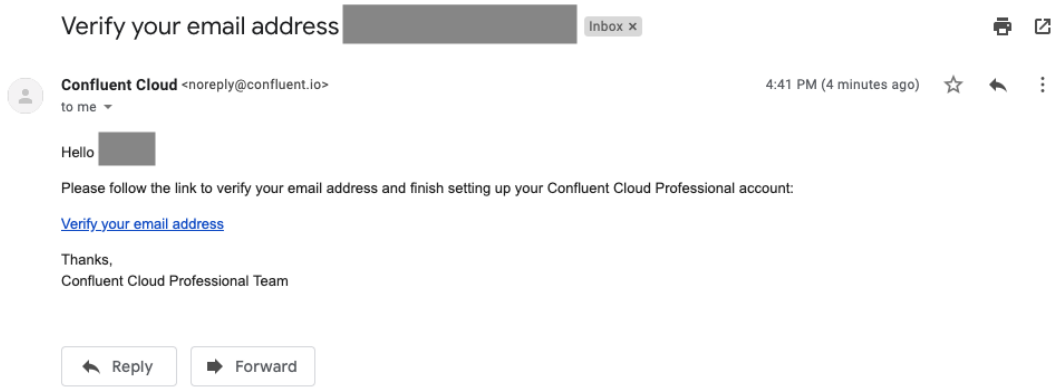
Getting Started

1. Navigate to Confluent Cloud portal at the URL https://www.confluent.io/confluent-cloud/.
2. Click the link **See Pricing** to get an idea about the cost of running a simple cluster either on GCP or AWS.
3. Sign up for an account by clicking the button **SIGN UP**.
4. Fill in the signup form:



and hit **Submit**

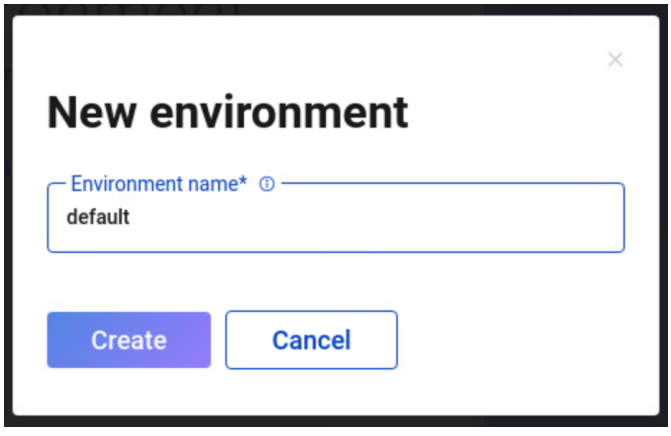
5. An email will be sent to you. Open it and click the link to confirm your email address:



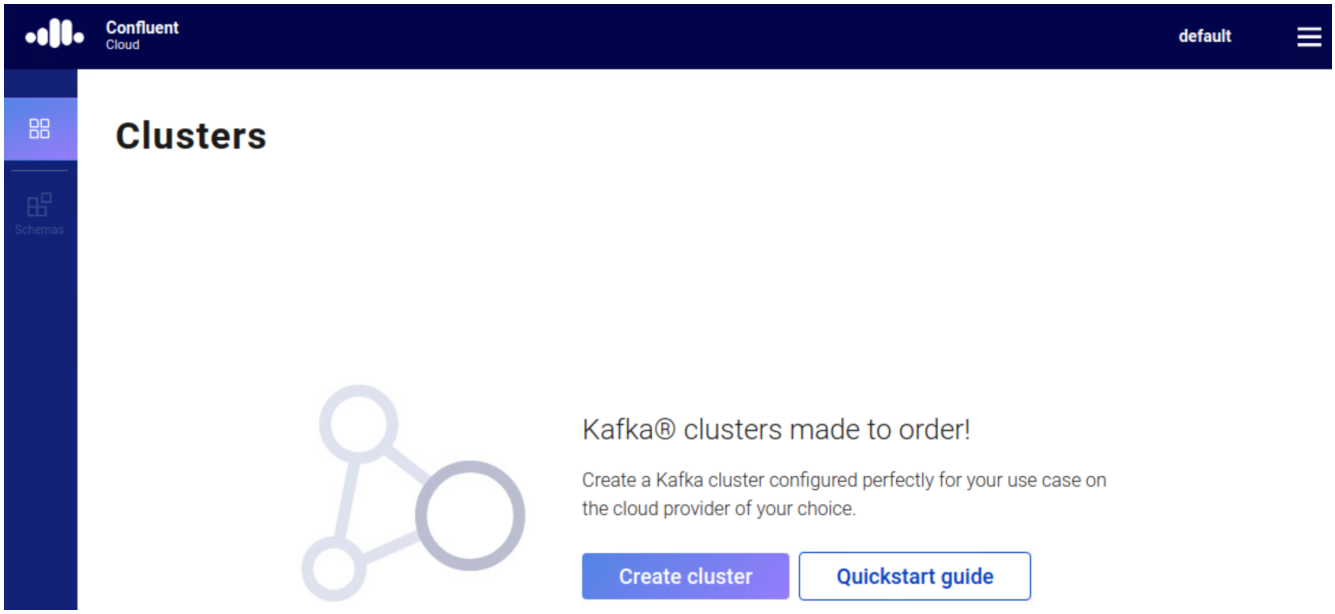
6. Upon clicking the above link you will be redirected to the Confluent Cloud home page and greeted by a screen similar to this:

If you login the first time, then there will be no environment defined.

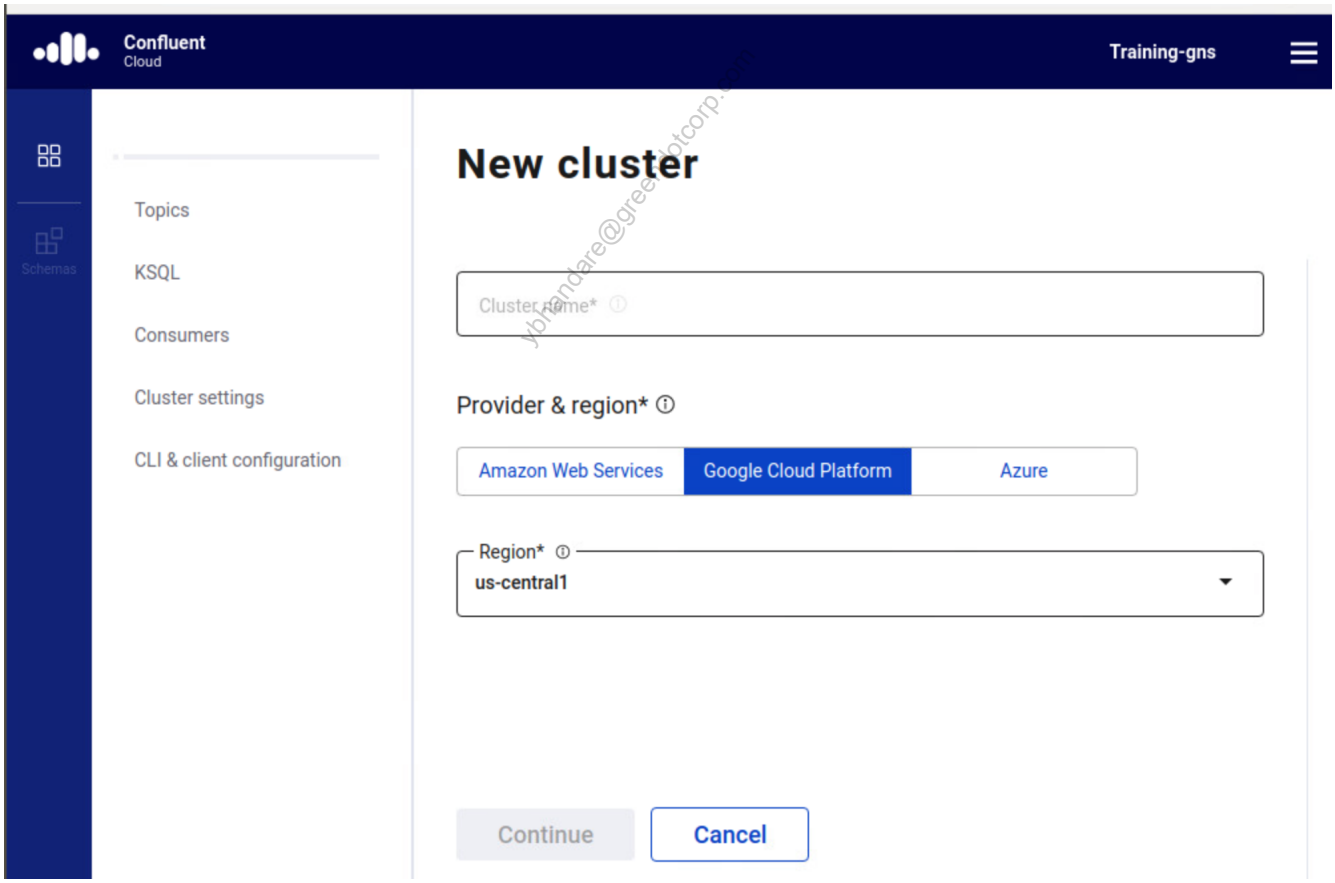
7. Create a new environment by clicking on **+ Add environment**.
8. Enter a name for the new environment such as `default` and then click **Create**:



9. In this form click the button **Create cluster** to create a first cluster:



10. Fill out the **New cluster** form:



You can create clusters on Azure, GCP or AWS:

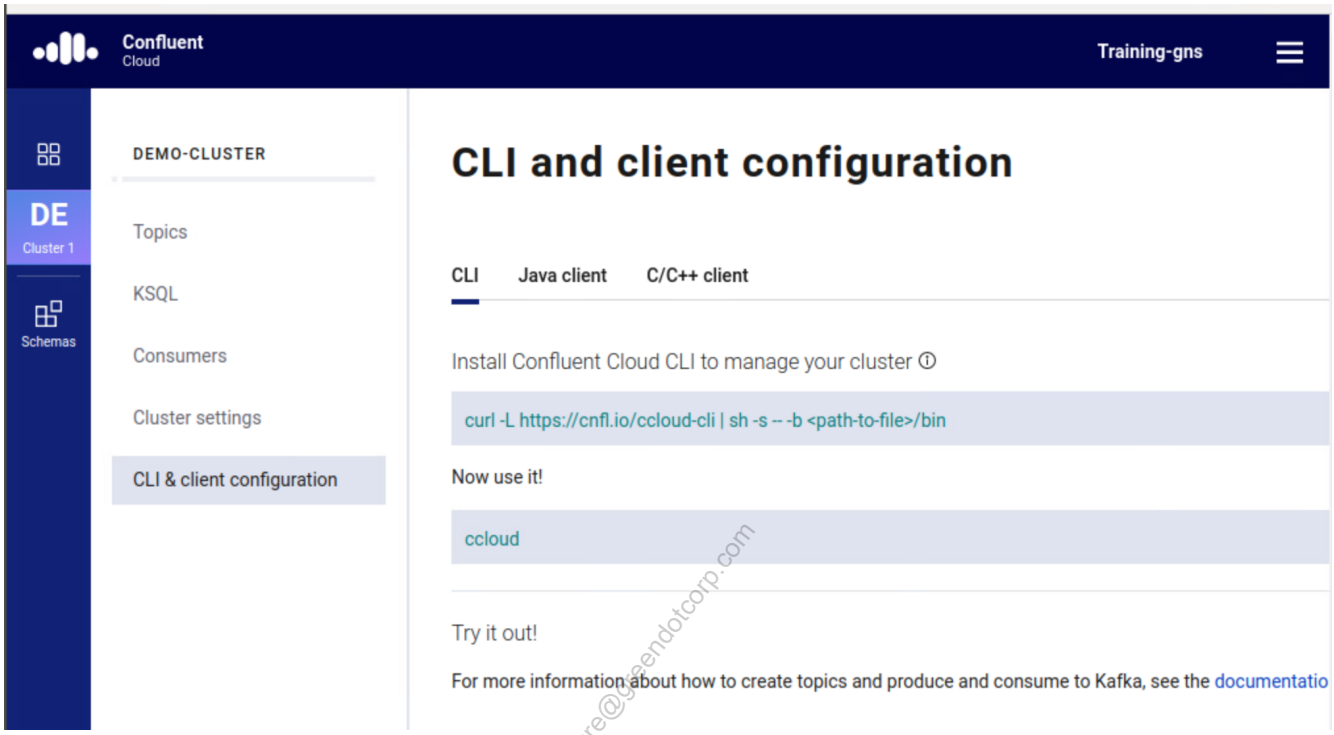
a. name the cluster `demo-cluster`,

b. selected GCP as **Provider**

c. chosen the **Region** closest to you, e.g. `us-central1`

11. Click **Continue**. You will be asked to enter your credit card info for payment of the resources consumed

12. Once your credit card has been validated you're redirected to the screen **CLI and client configuration**:



13. Follow the instructions on the screen and install the Confluent Cloud CLI.

a. If you're on a Mac e.g. use `/usr/local` where it says `<path-to-file>`

b. On Linux use this command:

```
$ curl -L https://cnfl.io/ccloud-cli | sudo sh -s -- -b /usr/bin
```

Note the `sudo` in the command!

14. Login with your Confluent Cloud CLI

```
$ ccloud login
Enter your Confluent Cloud credentials:
Email: <your email>
Password: <your password>

Logged in as <your email>
Using environment t182 ("default")
```

Use the same login credentials you used to login to the CCloud UI.

You may also get a message such as `Updates are available for ccloud`.... You can run `ccloud update` after you successfully logged in to update your tool.

15. List your environments:

```
$ ccloud environment list
    Id    |         Name
+---------+----------------------+
  * t182  | default
    t2841 | prod
    t2987 | staging
...
```

If you don't have an environment yet, you can create one e.g. named `demo-environment` via:

```
$ ccloud environment create demo-environment
```

16. Select the environment you want to work with:

```
$ ccloud environment use <Enviornment ID>
```

Use the appropriate environment ID instead of `<Enviornment ID>`

17. Now you can list your clusters in the selected environment:

```
$ ccloud kafka cluster list
      Id      | Name  | Provider |    Region    | Durability | Status
+-------------+-------+----------+--------------+------------+--------+
    lkc-436k2 | Alpha | gcp      | europe-west1 | LOW        | UP
```

To create a new cluster at this time you have to use the CCloud UI

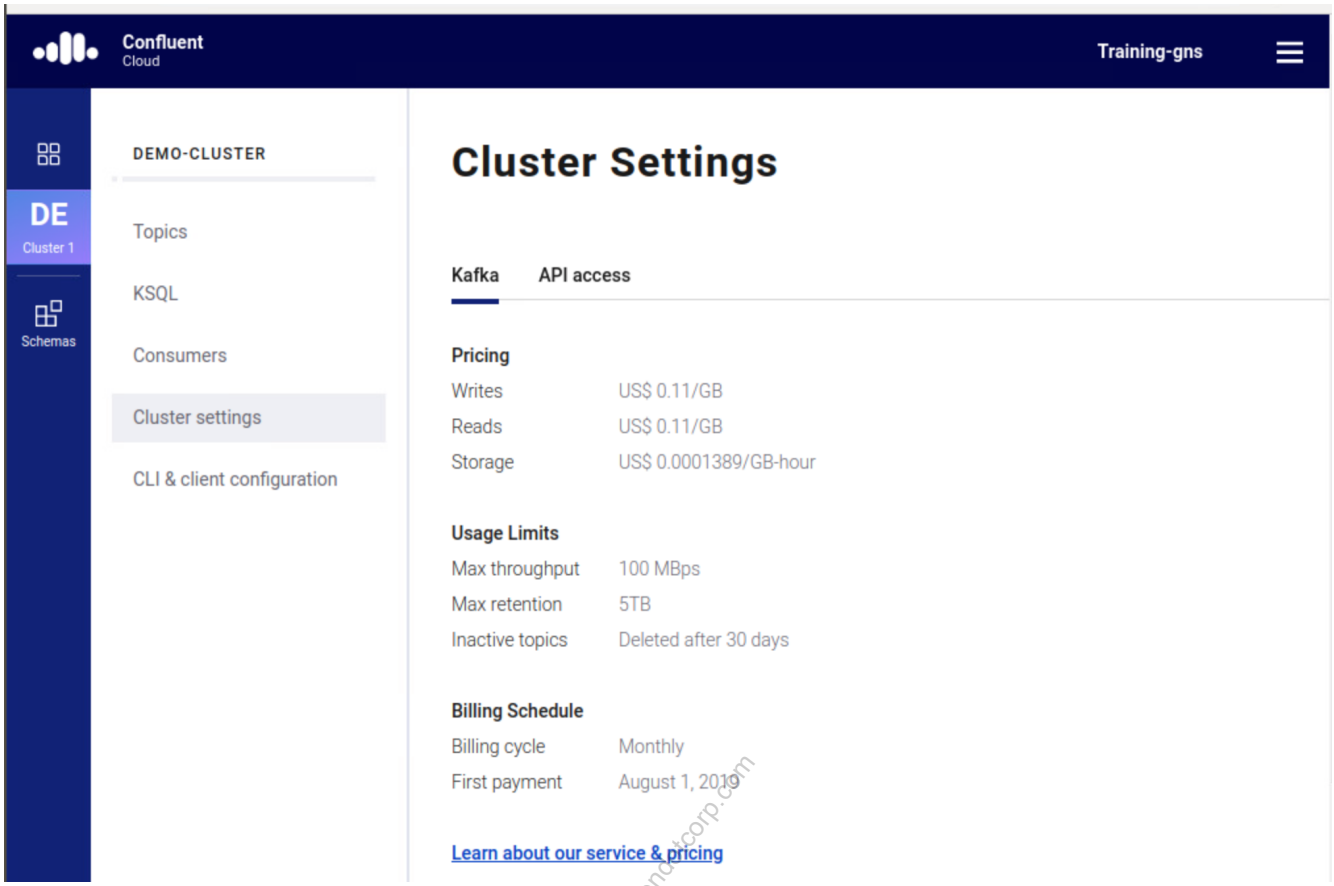
18. To use a cluster enter:

```
$ ccloud kafka cluster use <cluster ID>
```

Where `<cluster ID>` is the `Id` of the desired cluster.

A file `~/.ccloud/config.json` will be created which contains all the necessary configuration information collected by the CLI so far. OPTIONAL: have a look into its content.

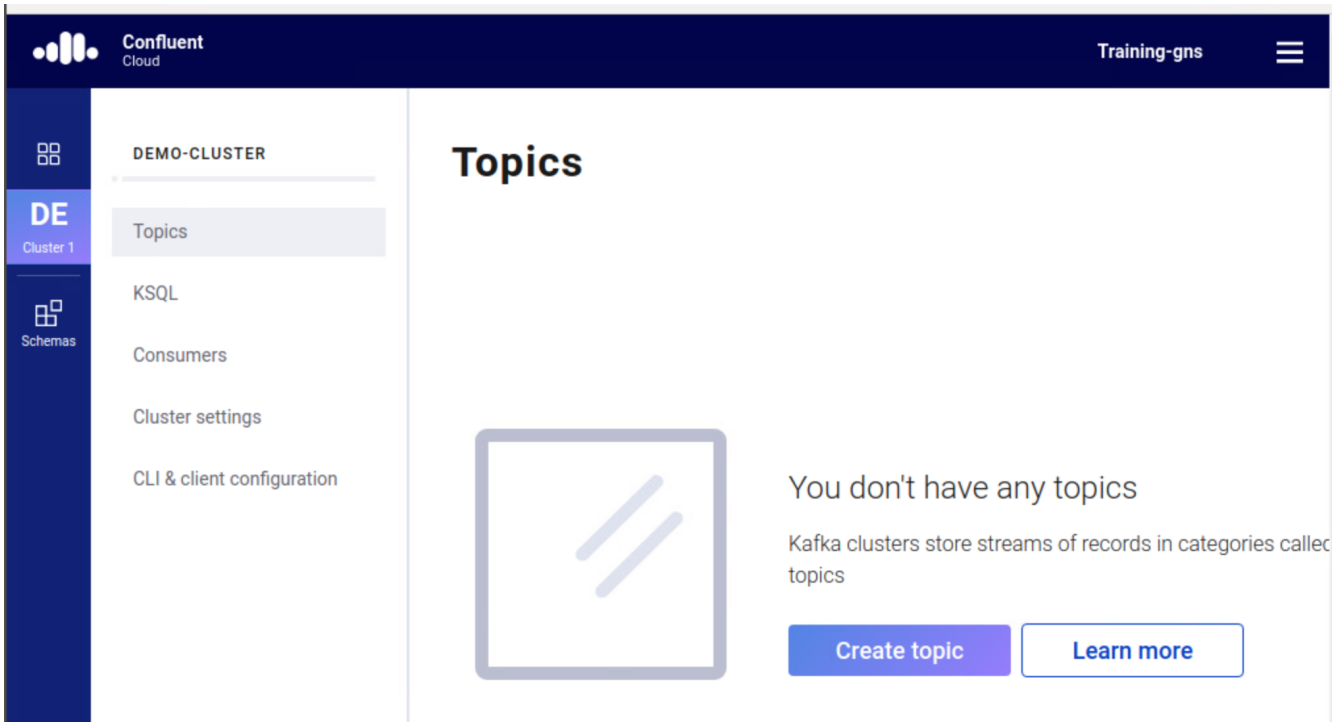
19. You can now explore your Confluent Cloud account. Navigate to **Cluster settings** and you should see something like this (make sure **Cluster 1** is selected in the blue bar on the left):



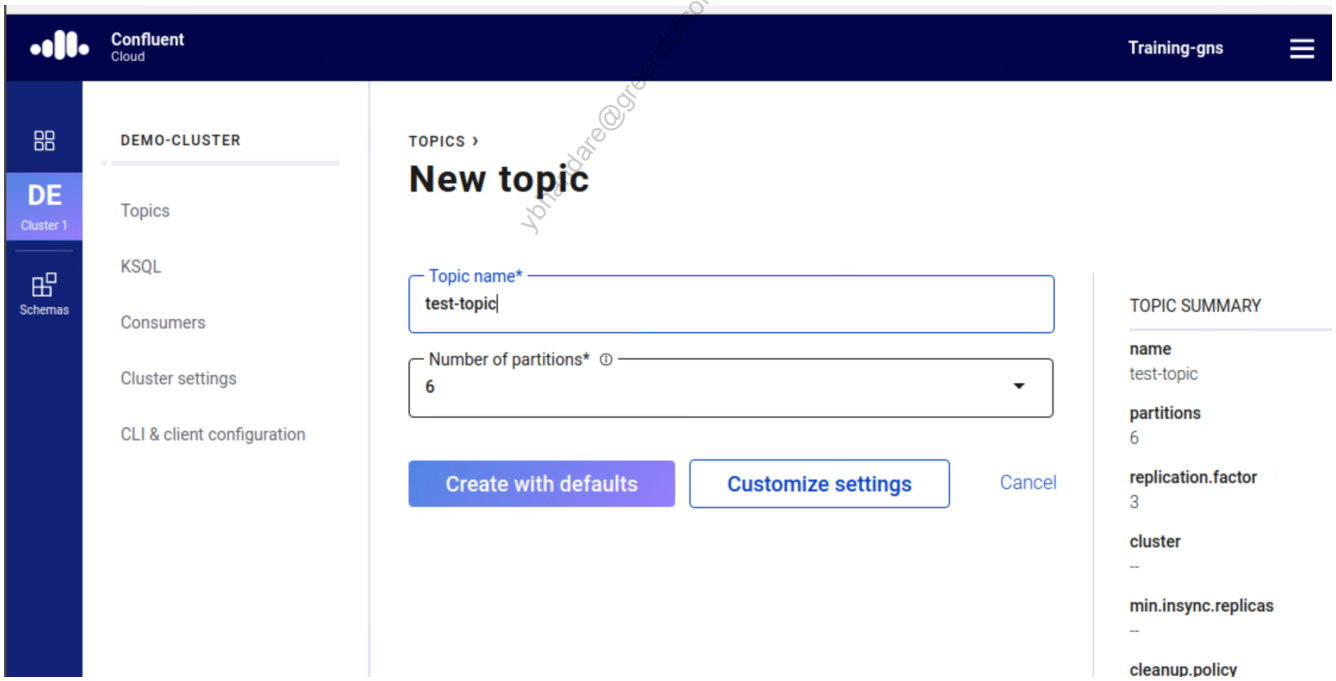
Click on the link **Learn about our service & pricing** to get more info about your expected cost.

Producing Data

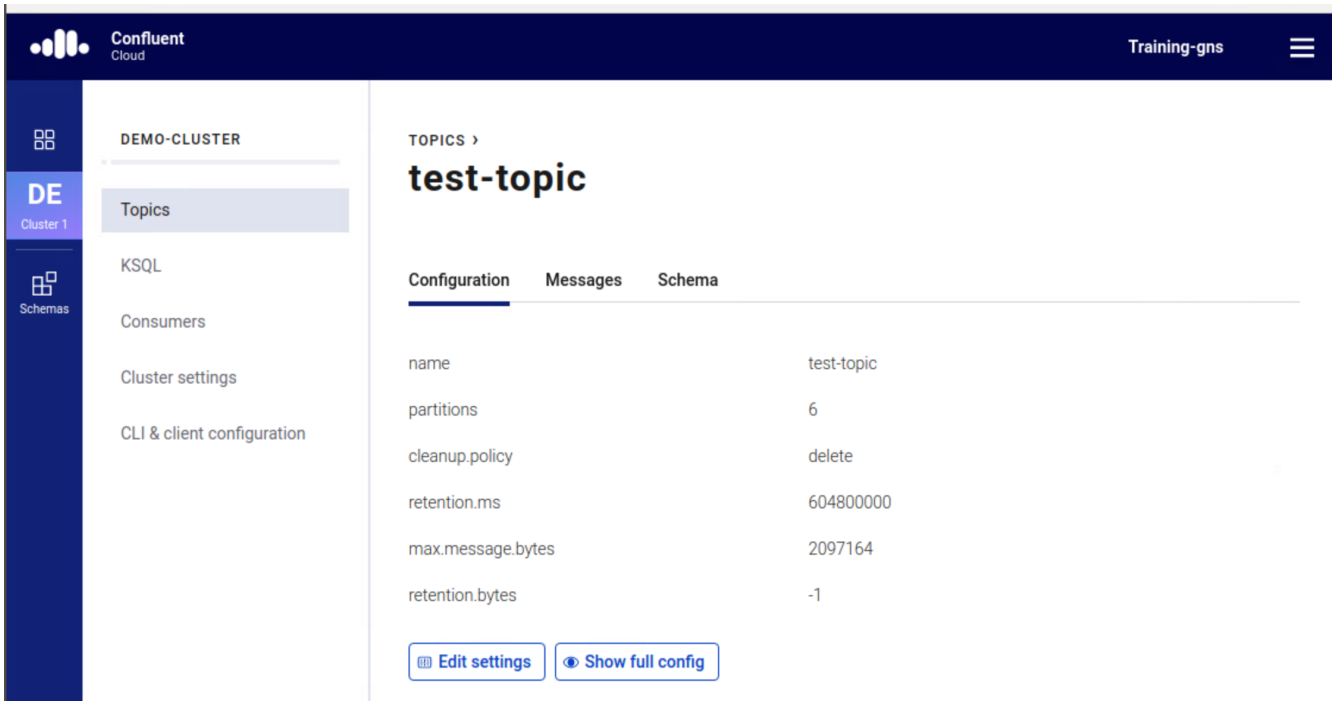
1. Let's first create a topic in Confluent Cloud. Navigate to **Topics** and click **Create topic**:



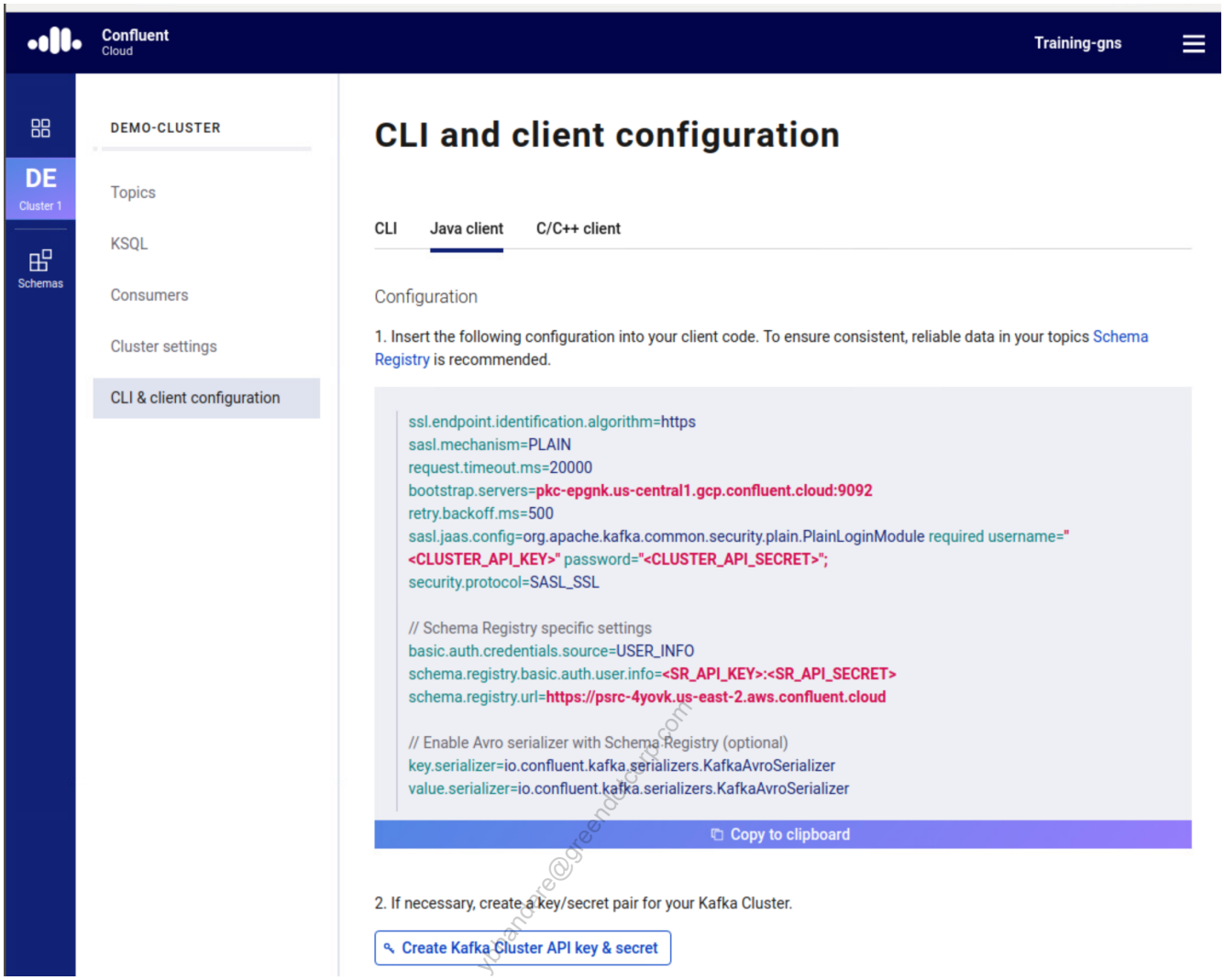
2. Enter a topic name such as `test-topic` and the number of partitions (e.g. 6):



and hit **Create with defaults**. You will see a confirmation screen similar to this:



3. In the UI navigate to **CLI & client configuration** and then select the **Java client** tab:



4. We first need to create a **Cluster API key & secret**. Click the button **Create Kafka Cluster API key & secret** to create one. Notice how the configuration settings in the gray block are updated with your key and secret.
5. Click **Copy to clipboard** to copy those settings.
6. In your terminal navigate to the `ccloud` folder:

```
$ cd ~/confluent-dev/labs/ccloud
```

7. In this folder create a file `client.properties` and paste the settings you just copied to the clipboard before into this file. It should look similar to this:

```
ssl.endpoint.identification.algorithm=https
sasl.mechanism=PLAIN
request.timeout.ms=20000
bootstrap.servers=<YOUR_BOOTSTRAP_SERVER>
retry.backoff.ms=500
sasl.jaas.config=org.apache.kafka.common.security.plain.PlainLoginModule required \
    username="<CLUSTER_API_KEY>" password="<CLUSTER_API_SECRET>";
security.protocol=SASL_SSL
...
```

8. Let's define a few environment variables to simplify the following commands:

```
$ export YOUR_BOOTSTRAP_SERVER=<YOUR_BOOTSTRAP_SERVER> &&
  export CLUSTER_API_KEY=<CLUSTER_API_KEY> &&
  export CLUSTER_API_SECRET=<CLUSTER_API_SECRET>
```

   Use the same values as in the pervious step!

9. Run the `kafka-console-producer` tool:

```
$ kafka-console-producer \
     --broker-list ${YOUR_BOOTSTRAP_SERVER} \
     --producer.config client.properties \
     --topic test-topic
>
```


The producer may output some warnings about unknown properties such as `basic.auth.credentials.source`. You can safely ignore those warning.

10. Enter some data:

```
>test item
>second item
>third item
>I love Kafka
```

    and press `CTRL-d` to exit the producer

11. To test if the data has been written to our hosted cluster we use the tool `kafka-console-consumer`:

```
$ kafka-console-consumer \
     --group demo-consumer-group \
     --bootstrap-server ${YOUR_BOOTSTRAP_SERVER} \
     --consumer.config client.properties \
     --topic test-topic \
     --from-beginning
```

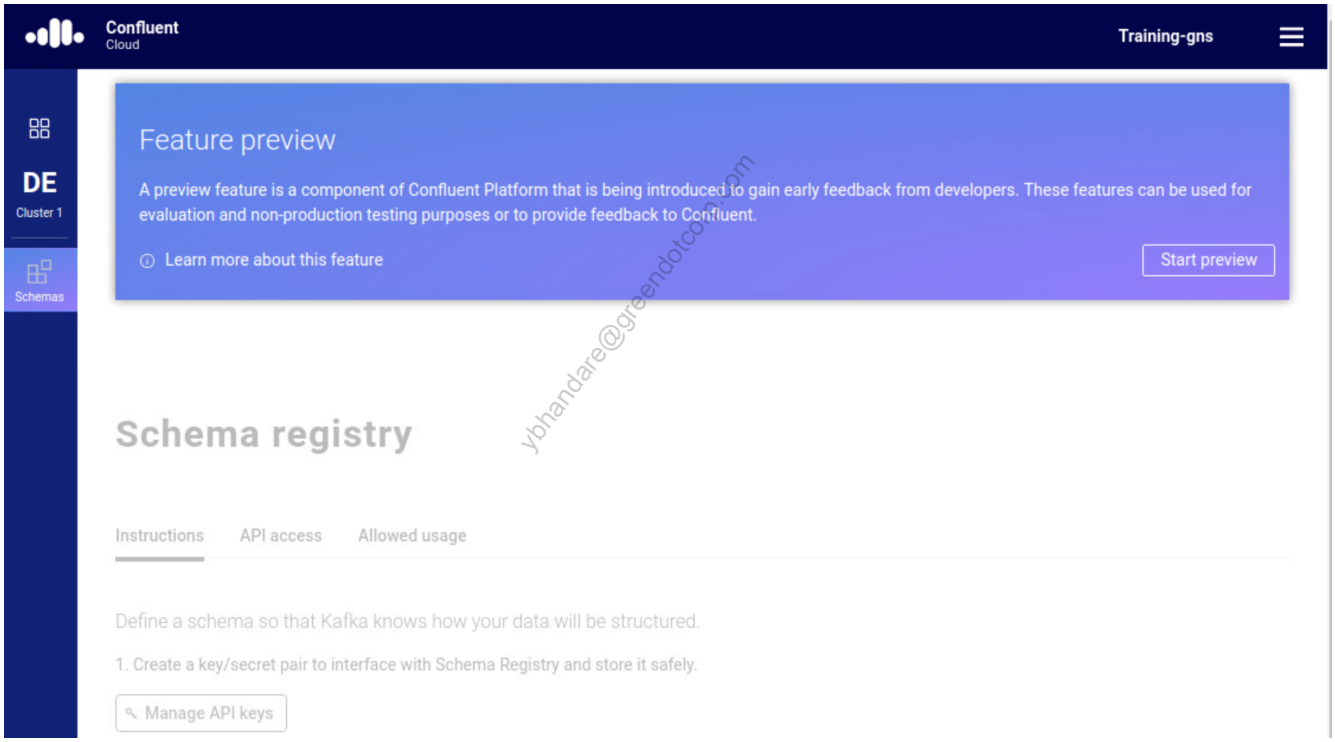
which should result in something like this:

```
test item
second item
I love Kafka
third item
```

the ordering is non-deterministic due to the fact that we have no key and 6 partitions. Also note that re-running the console consumer will not show the original messages again, even with `--from-beginning`, because we are explicitly setting a consumer group id

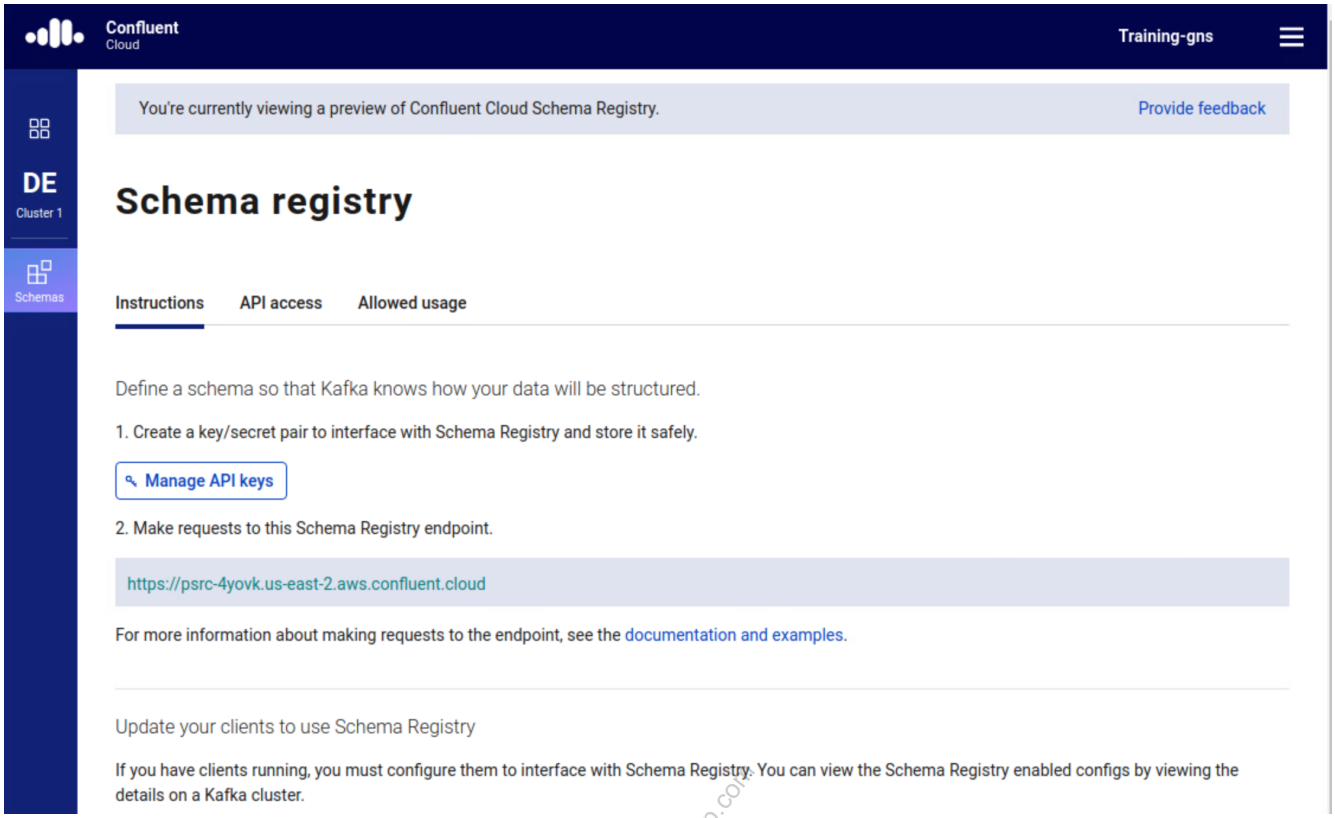
Using the Schema Registry

1. Select **Schemas** in the blue navigation bar on the left:



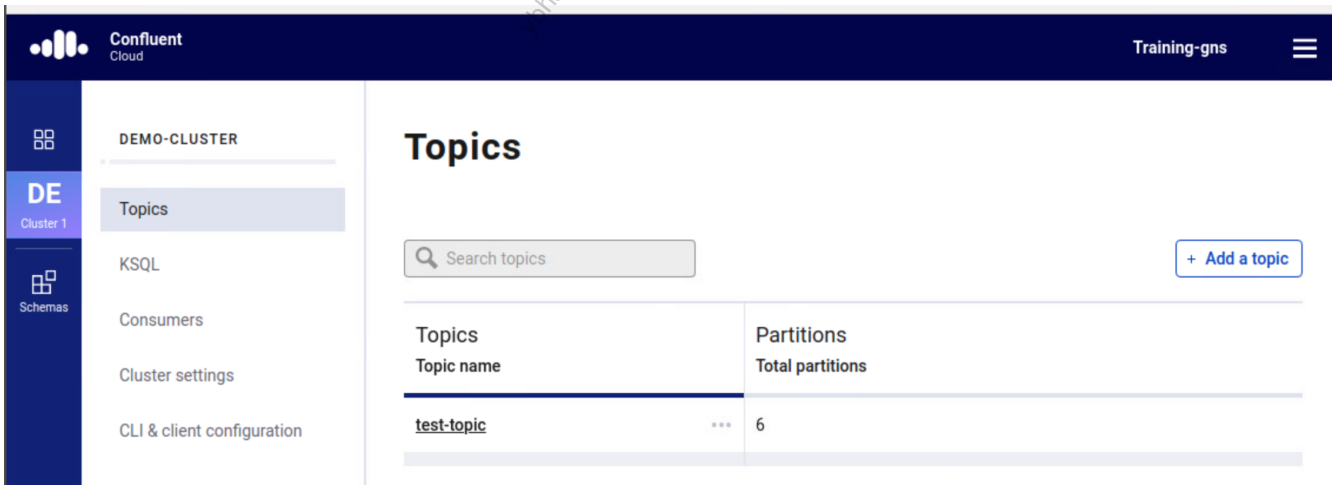
   At the time of writing hosted Schema Registry is still in preview.

2. Click the button **Start preview** to start your preview of the Schema Registry:

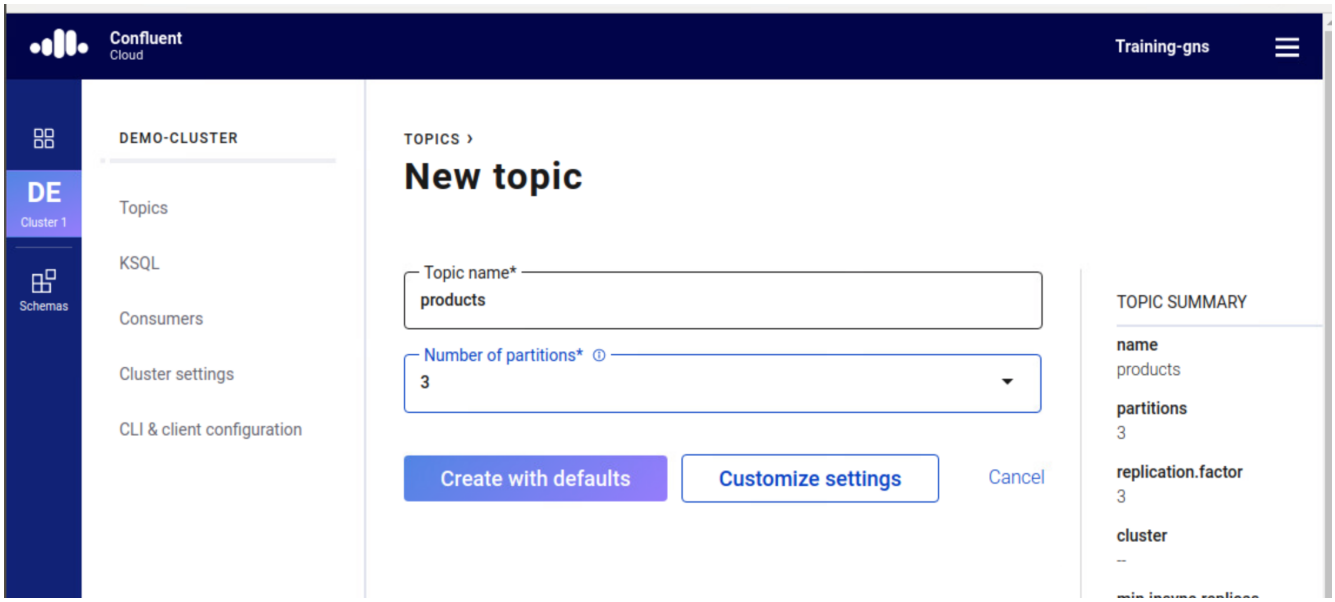


Currently there is no schema defined for topic `products`.

3. Let's create a new topic in Confluent Cloud. On Confluent Cloud navigate to **Topics** and hit the button **+ Create topic**:

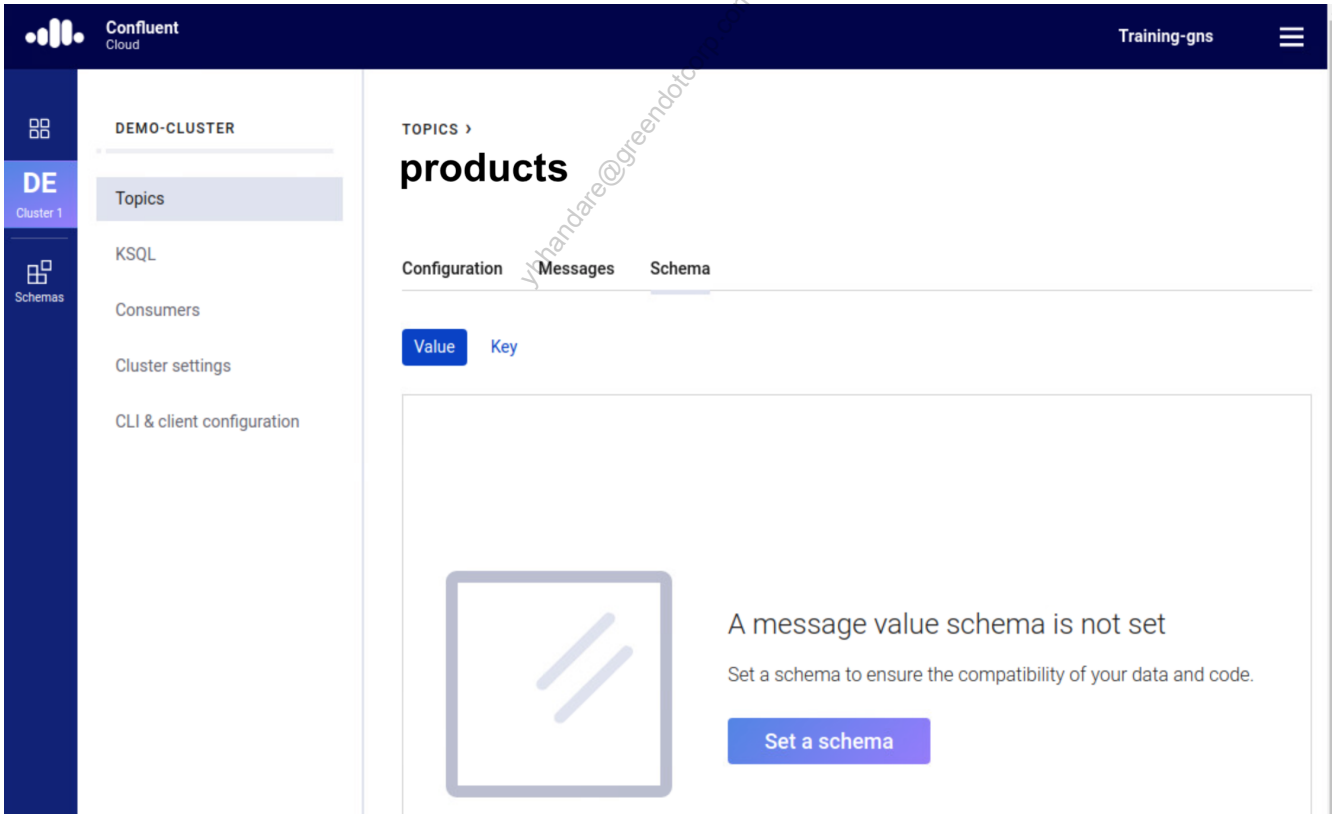


4. Call the topic `products` and set the number of partitions to 3:

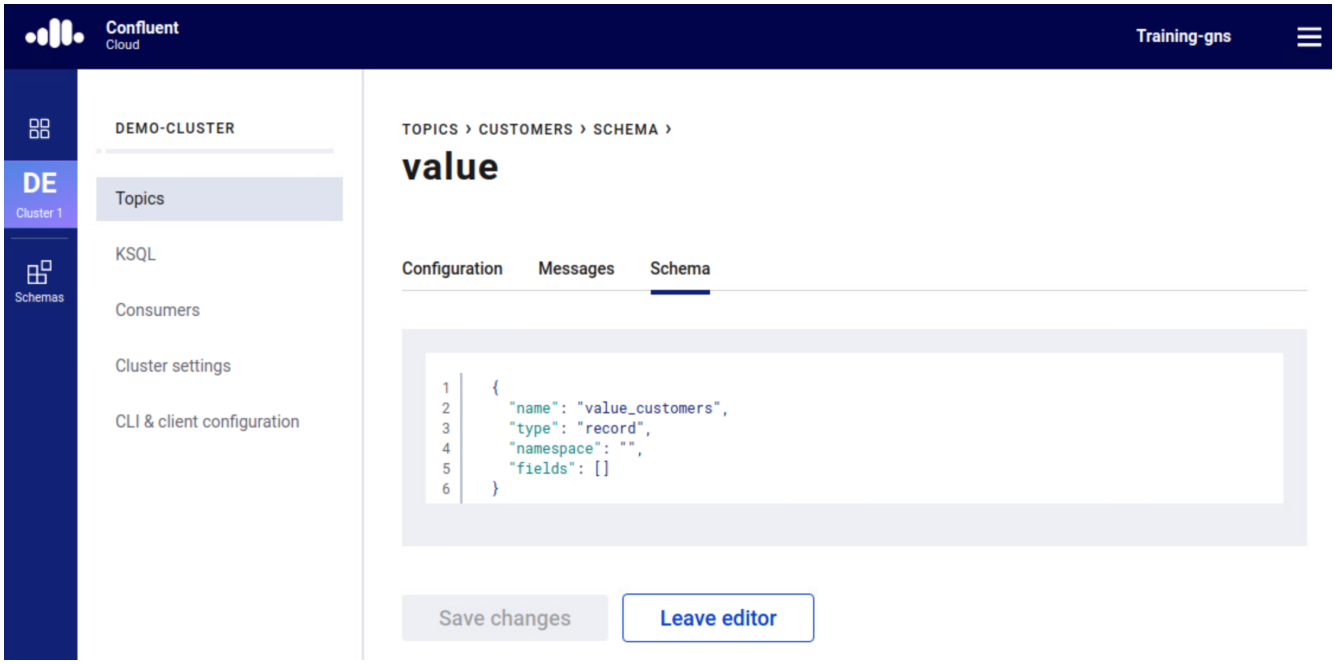


Then click **Create with defaults**.

5. Navigate to the **SCHEMA** tab:



6. Click the button **Set a schema**:



7. Modify the schema until it looks like this and then click **Save changes**:

```
{
  "type": "record",
  "name": "products",
  "namespace": "acme.com",
  "fields": [
    { "name": "id" "type": "int"},
    { "name": "name" "type": "string"},
    { "name": "unit_price" "type": "float"},
  ]
}
```

8. Now we need to create an API key/secret pair for our Schema Registry. Navigate to **CLI & client configuration** and scroll down until you see the button **Create Schema Registry API key & secret**
9. Click the button **Create Schema Registry API key & secret** to create a key/secret pair. Notice how your configuration gets updated
10. Edit the file `client.properties` you created earlier and add the necessary configuration for Schema Registry:

```
// Schema Registry specific settings
basic.auth.credentials.source=USER_INFO
schema.registry.basic.auth.user.info=<SR_API_KEY>:<SR_API_SECRET>
schema.registry.url=<YOUR_SR_URL>

// Enable Avro serializer with Schema Registry (optional)
key.serializer=io.confluent.kafka.serializers.KafkaAvroSerializer
value.serializer=io.confluent.kafka.serializers.KafkaAvroSerializer
```

Replace <SR_API_KEY> and <SR_API_SECRET> with your Schema Registry API Key and Secret. Also replace

<YOUR_SR_URL> with the the URL to your Schema Registry instance in Confluent Cloud

11. Let's define a few environment variables to simplify the following commands:

```
$ export SR_URL=<YOUR_SR_URL> &&
  export SR_KEY=<YOUR_SR_KEY> &&
  export SR_SECRET=<YOUR_SR_SECRET>
```

12. Let's define a schema for the record value:

```
$ PRODUCT_SCHEMA='{
    "type":"record",
    "namespace": "acme.com",
    "name":"product",
    "fields":[{"name":"id",         "type":"int"},
              {"name":"name",       "type":"string"},
              {"name":"unit_price","type":"float"}]
  }'
```

13. then we run the `kafka-avro-console-producer` with this schema:

```
$ kafka-avro-console-producer \
    --broker-list ${YOUR_BOOTSTRAP_SERVER} \
    --property basic.auth.credentials.source=USER_INFO \
    --property schema.registry.basic.auth.user.info=${SR_KEY}:${SR_SECRET} \
    --property schema.registry.url=${SR_URL} \
    --producer.config client.properties \
    --topic products \
    --property value.schema="${PRODUCT_SCHEMA}"
```


Due to a known but and although we're using the config file `client.properties` which has the credentials, we still need to provide them as `--property` to the tool `kafka-avro-console-producer`. The same applies to the `kafka-avro-console-consumer` below…
Also note that the `kafka-avro-console-producer` and `kafka-avro-console-consumer` generate a log of warnings about unrecognized properties, which you can safely ignore.

Alternatively to providing the schema directly via e.g. an environment variable to the tool one can also add the schema to a file, e.g. `product-schema.json` and then use this syntax instead:

```
--property value.schema="$(< product-schema.json)"
```

14. Enter a few values such as:

```
{ "id":1, "name": "apples", "unit_price": 1.25 }
{ "id":2, "name": "pears", "unit_price": 2.10 }
{ "id":3, "name": "apricots", "unit_price": 3.75 }
```

15. Exit the producer by pressing `CTRL-d`
16. Now lets try to read the Avro records in the topic `products` using the `kafka-avro-console-consumer` tool:

```
$ kafka-avro-console-consumer \
    --bootstrap-server ${YOUR_BOOTSTRAP_SERVER} \
    --property basic.auth.credentials.source=USER_INFO \
    --property schema.registry.basic.auth.user.info={SR_KEY}:${SR_SECRET} \
    --property schema.registry.url=${SR_URL} \
    --consumer.config client.properties \
    --from-beginning \
    --topic products

{"id":1,"name":"apples","unit_price":1.25}
{"id":2,"name":"pears","unit_price":2.1}
{"id":3,"name":"apricots","unit_price":3.75}
```

observe the values entered above output by the consumer.

17. Exit the consumer by pressing `CTRL-c`

Accessing the Schema Registry via REST API

We can also use the REST API of the Schema Registry for management tasks. Since the SR is protected we need to provide the necessary credentials in our REST calls.

1. Let's define a schema for the record value. Create a file `product.json` with the following content:

```
{
    "schema":
    "{
        \"namespace\": \"app\",
        \"type\": \"record\",
        \"name\": \"product\",
        \"fields\":
        [
            { \"name\": \"id\", \"type\": \"int\" },
            { \"name\": \"name\", \"type\": \"string\" },
            { \"name\": \"unit_price\", \"type\": \"double\" }
        ]
    }"
}
```

2. To register the above as a new schema in the registry use this command:

```
$ curl -X POST \
    -u "${SR_KEY}:${SR_SECRET}" \
    -H "Content-Type: application/vnd.schemaregistry.v1+json" \
    --data @product.json \
    "${SR_URL}/subjects/product-value/versions"

{"id":100001}
```

A different ID may be returned in your case... Why?

3. To list all registered subjects on your SR execute:

```
$ curl -u "${SR_KEY}:${SR_SECRET}" "${SR_URL}/subjects"
["products-value"]
```

You may see more schemas in the output. Which ones?

4. Get the latests registered schema for a particular subject:

```
$ curl -u "${SR_KEY}:${SR_SECRET}" \
    "${SR_URL}/subjects/products-value/versions/latest" | jq .schema

"{\"type\":\"record\",\"name\":\"test\",\"namespace\":\"app\",\"fields\":[{\"name\":\"id\
",\"type\":\"int\"},{\"name\":\"name\",\"type\":\"string\"},{\"name\":\"unit_price\",\"ty
pe\":\"double\"}]}"
```


5. Deleting an existing schema, e.g. `product-value`:

```
$ curl -X DELETE \
    -u "${SR_KEY}:${SR_SECRET}" \
    "${SR_URL}/subjects/products-value"

[1]
```

This normally should only be done when a topic needs to be recycled or in a development environment.

Using a Java Avro Producer with Confluent Cloud

1. Navigate to the project folder for the producer:

```
$ cd ~/confluent-dev/labs/ccloud/producer
```

2. Run a gradle build to generate POJOs from the Avro schemas, then launch VS Code.

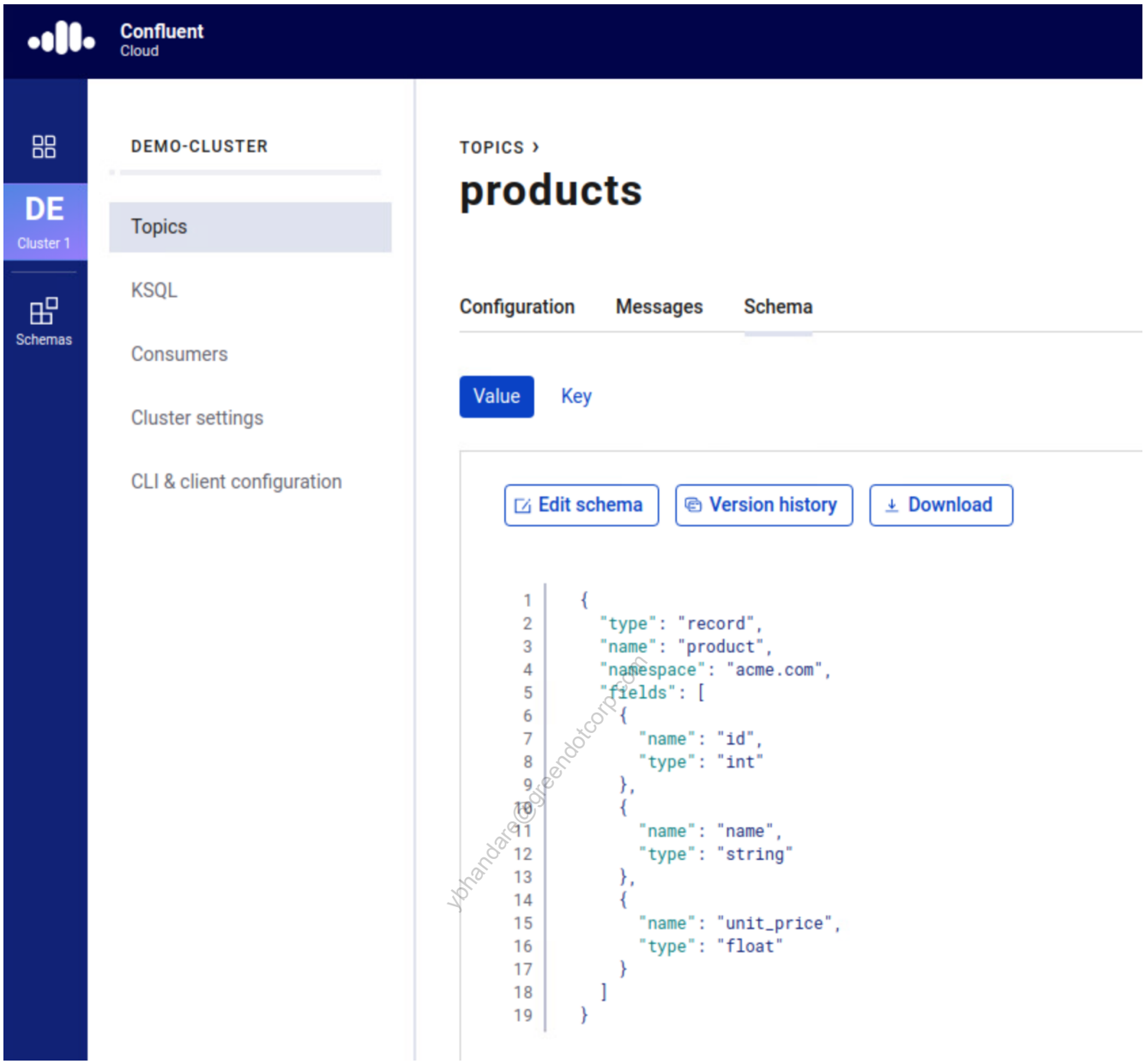
```
$ gradle build
$ code .
```

3. Locate the file `src/main/avro/product_value.avsc` and analyze the schema defined therein
4. Open the file `src/main/java/clients/ProductProducer.java` and analyze its code:
   a. In the configuration block replace the placeholders with the specific values for your Confluent Cloud cluster and Schema Registry
   b. Notice how we create one instance of `ProductValue` and use the producer to write it to the Kafka cluster in Confluent Cloud
5. Run the producer by selecting the menu **Debug** → **Start Debugging** in VS Code.

   You should see something like this:

```
*** Starting prod-producer ***
...
### Stopping prod-producer ###
```

6. Head over to Confluent Cloud and navigate to **Topics**. Select the `products` topic and then the **SCHEMA** tab. You should see something like:



No surprise here, this is exactly what we expected

Using a Java Avro consumer with Confluent Cloud

1. Navigate to the project folder for the producer:

```
$ cd ~/confluent-dev/labs/ccloud/consumer
```

2. Run a gradle build to generate POJOs from the Avro schemas, then launch VS Code.

```
$ gradle build
$ code .
```

3. Locate the file `src/main/avro/product_value.avsc` and analyze the schema defined therein. It's the same we used for the producer
4. Open the file `src/main/java/clients/ProductConsumer.java` and analyze its code
5. In the configuration block replace the placeholders with the specific values for your Confluent Cloud cluster and Schema Registry
6. Run the consumer by selecting the menu **Debug** → **Start Debugging** in VS Code.

   you should see an output like this:

```
*** Starting Product Consumer Avro ***
...
offset = 0, key = 1, value = {"id": 1, "name": "apples", "unit_price": 1.25}
```

7. Stop the consumer with `CTRL-C`

Editing Schemas in the Web UI

You can also create new and modify existing schemas in the Web UI.

1. In Confluent Cloud navigate to **Topics** and create a new topic `orders`
2. Once the topic is created navigate to the tab **SCHEMA**
3. Make sure the **Value** tab is selected and then click the button **Set a schema**
4. In the editor modify the template to look like this:

```
{
  "type": "record",
  "name": "value_orders",
  "namespace": "acme.com",
  "fields": [
    {
      "name": "customer_id",
      "type": "int"
    },
    {
      "name": "order_no",
      "type": "string"
    },
    {
      "name": "order_amount",
      "type": "float"
    }
  ]
}
```

and click **Save changes**

5. Try to create a BACKWARD compatible new version of the schema by adding a field like this:

```
{
    "name": "item_count",
    "type": "int",
    "default": 0
},
```


you have to add a `default` value for backwards compatibility.

You can also use `null` as default value as follows:

```
{
    "name": "item_count",
    "type": ["null", "int"],
    "default": null
},
```

If you need to delete a topic, then at this time you have to use the `ccloud` command line tool, e.g.:

```
$ ccloud topic delete <topic-name>
```

OPTIONAL: Using the Cloud CLI

Here we are using the new Confluent Cloud CLI (V2) to manage our cluster. It is currently in preview

1. Install the tool `ccloud`.
2. Once installed, login to your cloud account

```
$ ccloud login

Enter your Confluent Cloud credentials:
Email:: <YOUR_EMAIL>
Password: <YOUR_PASSWORD>

Logged in as xyz@confluent.io
Using environment <Enviornment ID> ("Training")
```

3. List all define environments:

```
$ ccloud environment list
```

4. Select your environment:

```
$ ccloud environment use <YOUR_ENVIRONMENT>
```

5. List the clusters in your active environment:

```
$ ccloud kafka cluster list

      Id      | Name  | Provider |    Region    | Durability | Status
+-------------+-------+----------+--------------+------------+--------+
  * lkc-lovz9 | Alpha | gcp      | europe-west3 | LOW        | UP
```

6. Select the cluster you want to work with:

```
$ ccloud kafka cluster use <CLUSTER_ID>
```

7. Create an API key and secret pair:

```
$ ccloud api-key create --cluster lkc-lovz9 --description Demo

Please save the API Key and Secret. THIS IS THE ONLY CHANCE YOU HAVE!
+---------+------------------------------------------------------------------+
| API Key | PNKRQOOZT2NYYWDF                                                 |
| Secret  | NKgycWpK0HlWNV0jLXMSrOktuw9O/BentUQsylkBnirFH0bzZkbeE3y9YQqAeSCz |
+---------+------------------------------------------------------------------+
```

8. Now list your api keys:

```
$ ccloud api-key list

ccloud api-key list
         Key         | Owner | Description
+--------------------+-------+-------------+
    QPH5ZYWQY5QYTLVE |  5209 | Other key
    PNKRQOOZT2NYYWDF |  5209 | Demo
```

9. Define the key you want to work with:

```
$ ccloud api-key use <KEY>
```

10. Let's create a topic:

```
$ ccloud kafka topic create customers --partitions 3
```

11. We can describe the topic just created:

```
$ ccloud kafka topic describe customers

Topic: customers PartitionCount: 3 ReplicationFactor: 3
    Topic    | Partition | Leader | Replicas |   ISR
+-----------+-----------+--------+----------+---------+
  customers |         0 |      6 | [6 2 0]  | [6 2 0]
  customers |         1 |      4 | [4 6 8]  | [4 6 8]
  customers |         2 |      2 | [2 9 4]  | [2 9 4]

Configuration

                   Name                   |        Value
+-----------------------------------------+---------------------+
  compression.type                         | producer
  leader.replication.throttled.replicas    |
  min.insync.replicas                      |                   2
  message.downconversion.enable            | true
..
```

12. Now let's produce some data into topic `customers`:

```
$ ccloud kafka topic produce customers

Starting Kafka Producer. ^C to exit
1:{"id":1, "name": "Joe"}
2:{"id":2, "name": "Ann"}
3:{"id":3, "name": "Sue"}
^C
```

13. Consuming is equally straight forward:

```
$ ccloud kafka topic consume customers --from-beginning

Starting Kafka Consumer. ^C to exit
{"id":1, "name": "Joe"}
{"id":2, "name": "Ann"}
{"id":3, "name": "Sue"}
^C
```

14. Finally to delete the topic use:

```
$ ccloud kafka topic delete customers
```


Cleanup

1. Delete your cluster in Confluent Cloud to avoid unexpected cost:
    a. Select you cluster **Cluster 1**
    b. Select **Cluster settings** and then click the button **Change settings**
    c. Click the **Delete** link right next to the **Cancel** button

Conclusion

In this exercise you have used Confluent Cloud to host a fully managed Kafka cluster. We have manually created a topic and used the Confluent Cloud CLI to initialize our environment for access of the hosted cluster. Then we have used the Kafka console producer and consumer tools to write and consume data to and from the topic we created. Finally we also used a preview of the Schema Registry to manage an Avro schema we created on the fly.

STOP HERE. THIS IS THE END OF THE EXERCISE.

Appendix A: KSQL for Data Scientists & Data Engineers

KSQL for Data Scientists & Data Engineers

The use of Jupyter notebooks is very popular among data scientists. In this exercise you are going to use KSQL from within a Jupyter notebook, to analyze data originating from the Kafka topic `vehicle-positions`. This topic in turn is populated by a producer you have created in an earlier lab. The producer will consume the data from the **High-frequency Positioning** service of **digitransit**.

Prerequisites

1. Create the project folder, and navigate to it:

```
$ mkdir -p ~/confluent-dev/labs/data-science &&
  cd ~/confluent-dev/labs/data-science
```

2. Run the Kafka cluster, including Confluent Control Center:

```
$ docker-compose up -d
```

3. Create the topic `vehicle-positions` with this command:

```
$ kafka-topics \
    --create \
    --bootstrap-server kafka:9092 \
    --partitions 6 \
    --replication-factor 1 \
    --topic vehicle-positions
```

4. Run a Docker container with the Kafka producer from the lab **Creating a Kafka Producer**:

```
$ docker container run --rm -d \
    --name vp-producer \
    --hostname vp-producer \
    --net labs_confluent \
    cnfltraining/vp-producer:v2
```

If you were not able to finish the lab **Creating a Kafka Producer** then don't worry. The necessary image `cnfltraining/vp-producer:v2` will just be downloaded from our Confluent Training org on Docker Hub.

If you encounter problems with the producer, e.g. the MQTT datasource that the producer is relying on is down, then please use the Docker image `cnfltraining/vp-producer-fallback:v2` instead.

5. If not already installed (check with `pip3 --version`), then install `pip3`:

```
$ sudo apt update && sudo apt install python3-pip
```

6. Install **Jupyter**:

```
$ sudo -H pip3 install jupyter
```

7. Test that `jupyter` has been installed:

```
$ jupyter --version
4.5.0
```

8. Install the `ksql` Python library:

```
$ pip3 install ksql

Collecting ksql
  Downloading
https://files.pythonhosted.org/packages/72/35/a24d5cbcadde26b44cabfde0bcba49b639e77a73da8
094e387db7effba77/ksql-0.5.1.1.tar.gz
Collecting requests (from ksql)
...
Successfully built ksql
Installing collected packages: certifi, idna, chardet, requests, ksql
Successfully installed certifi-2019.3.9 chardet-3.0.4 idna-2.8 ksql-0.5.1.1 requests-
2.21.0
```


Analyzing the Vehicle Position Data

1. Start **Jupyter**:

```
$ jupyter notebook
```

And you should see something similar to this (shortened):

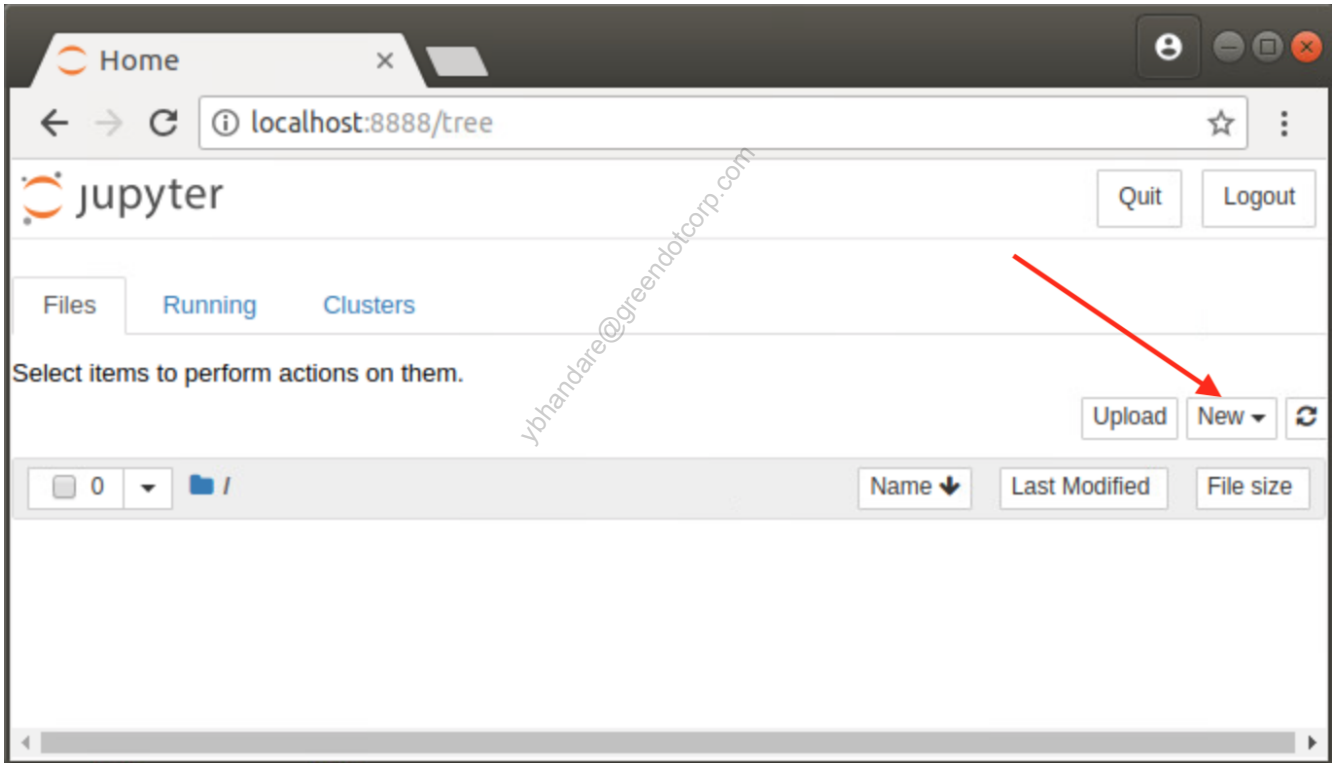
```
training@localhost:~/jupyter$ sudo jupyter notebook --allow-root
[I 07:37:31.592 NotebookApp] Serving notebooks from local directory: /home/training/jupyter
[I 07:37:31.592 NotebookApp] The Jupyter Notebook is running at:
[I 07:37:31.592 NotebookApp] http://localhost:8888/?token=3a4869e23c2a2fc1593d957b5efa09030de000ea014ea65b
[I 07:37:31.592 NotebookApp] Use Control-C to stop this server and shut down all kernels (twice to skip confirmation).
[C 07:37:31.600 NotebookApp]

    To access the notebook, open this file in a browser:
        file:///home/training/.local/share/jupyter/runtime/nbserver-5565-open.html
    Or copy and paste one of these URLs:
        http://localhost:8888/?token=3a4869e23c2a2fc1593d957b5efa09030de000ea014ea65b
```

Please note the URL you're asked to copy!

If the above command does not work, then try with:

```
$ sudo jupyter notebook --allow-root
```

2. Your browser should open automatically and you should see this:



3. Click **New** and select **Python 3**, to create a new Python 3 notebook.
4. In the first input box (grey area) of the notebook enter the following statement to load the `KSQLAPI` object from the `ksql` library:

```
from ksql import KSQLAPI
```

and hit `SHIFT-ENTER` to execute the command.

5. Use the KSQLAPI object to connect to the KSQL server with the following command:

```
client = KSQLAPI('http://localhost:8088')
```

Do not forget to hit SHIFT-ENTER to execute the command.

6. Next let's define a KSQL stream from the Kafka topic `vehicle-positions`:

```
client.create_stream(
    table_name='vpositions',
    columns_type=['''VP STRUCT<
        desi  STRING,
        dir  STRING,
        oper  INTEGER,
        veh  INTEGER,
        tst  STRING,
        tsi  BIGINT,
        spd  DOUBLE,
        hdg  INTEGER,
        lat  DOUBLE,
        long  DOUBLE,
        acc  DOUBLE,
        dl  INTEGER,
        odo  INTEGER,
        drst  INTEGER,
        oday  STRING,
        jrn  INTEGER,
        line  INTEGER,
        start  STRING,
        loc  STRING,
        stop  STRING,
        route  STRING,
        occu  INTEGER,
        seq  INTEGER>'''],
    topic='vehicle-positions',
    value_format='JSON')
```


and hit SHIFT-ENTER.

7. If you have followed the instructions exactly you should see this:

```
In [1]: from ksql import KSQLAPI

In [2]: client = KSQLAPI('http://localhost:8088')

In [3]: client.create_stream(
            table_name='vpositions',
            columns_type=['''VP STRUCT<
                desi  STRING,
                dir  STRING,
                oper  INTEGER,
                veh  INTEGER,
                tst  STRING,
                tsi  BIGINT,
                spd  DOUBLE,
                hdg  INTEGER,
                lat  DOUBLE,
                long  DOUBLE,
                acc  DOUBLE,
                dl  INTEGER,
                odo  INTEGER,
                drst  INTEGER,
                oday  STRING,
                jrn  INTEGER,
                line  INTEGER,
                start  STRING,
                loc  STRING,
                stop  STRING,
                route  STRING,
                occu  INTEGER,
                seq  INTEGER>'''],
            topic='vehicle-positions',
            value_format='JSON')
```

8. We can also use the function `ksql` to execute any KSQL statement such as describing the stream we just generated:

```
client.ksql('describe vpositions')
```

which results in the following output (shortened):

```
[{'@type': 'sourceDescription',
  'statementText': 'describe vpositions;',
  'sourceDescription': {'name': 'VPOSITIONS',
   'readQueries': [],
   'writeQueries': [],
   'fields': [{'name': 'ROWTIME',
     'schema': {'type': 'BIGINT', 'fields': None, 'memberSchema': None}},
    {'name': 'ROWKEY',
     'schema': {'type': 'STRING', 'fields': None, 'memberSchema': None}},
    {'name': 'VP',
     'schema': {'type': 'STRUCT',
      'fields': [{'name': 'DESI',
        'schema': {'type': 'STRING', 'fields': None, 'memberSchema': None}},
       {'name': 'DIR',
        'schema': {'type': 'STRING', 'fields': None, 'memberSchema': None}},
       {'name': 'OPER',
        'schema': {'type': 'INTEGER', 'fields': None, 'memberSchema': None}},
       {'name': 'VEH',
        'schema': {'type': 'INTEGER', 'fields': None, 'memberSchema': None}},
       ...
       {'name': 'SEQ',
        'schema': {'type': 'INTEGER', 'fields': None, 'memberSchema': None}}],
      'memberSchema': None}}],
   'type': 'STREAM',
   'key': '',
   'timestamp': '',
   'statistics': '',
   'errorStats': '',
   'extended': False,
   'format': 'JSON',
   'topic': 'vehicle-positions',
   'partitions': 0,
   'replication': 0}}]
```


9. Now let's execute a query:

```
result = client.query('select * from vpositions limit 5')
for item in result: print(item)
```

This command returns a generator. It can be printed e.g. by reading its values via `next(result)` or a `for` loop as above. After a short moment you should see a result similar to this:

```
{"row":{"columns":[1572517399119,"/hfp/v2/journey/ongoing/vp/bus/0017/000
41/1054/2/Itäkeskus(M)/12:27/1465148/5/60;24/28/27/06",{"DESI":"54","DIR":"2","OPER":17,"
VEH":41,"TST":"2019-10-
31T10:23:27.040Z","TSI":1572517407,"SPD":0.0,"HDG":150,"LAT":60.220539,"LONG":24.876
866,"ACC":0.0,"DL":240,"ODO":5,"DRST":0,"ODAY":"2019-10-
31","JRN":874,"LINE":71,"START":"12:27","LOC":"GPS","STOP":"1465148","ROUTE":"1054","OCCU
":0,"SEQ":null}]},"errorMessage":null,"finalMessage":null,"terminal":false}
{"row":{"columns":[1572517399119,"/hfp/v2/journey/ongoing/vp/bus/0018/00265/1064/1/Itä-
Pakila/12:25/1020123/5/60;24/19/74/13",{"DESI":"64","DIR":"1","OPER":18,"VEH":265,"TST":"
2019-10-
31T10:23:26.904Z","TSI":1572517406,"SPD":0.1,"HDG":177,"LAT":60.17133,"LONG":24.943242,"A
CC":0.0,"DL":120,"ODO":10,"DRST":1,"ODAY":"2019-10-
31","JRN":209,"LINE":80,"START":"12:25","LOC":"GPS","STOP":"1020123","ROUTE":"1064","OCCU
":0,"SEQ":null}]},"errorMessage":null,"finalMessage":null,"terminal":false}
{"row":{"columns":[1572517399131,"/hfp/v2/journey/ongoing/vp/tram/0040/00079/1001/2/Eira/
12:12/1220415/4/60;24/19/95/34",{"DESI":"1
","DIR":"2","OPER":40,"VEH":79,"TST":"2019-10-
31T10:23:27.033Z","TSI":1572517407,"SPD":7.27,"HDG":228,"LAT":60.19316,"LONG":24.954084,"
ACC":0.15,"DL":-58,"ODO":3469,"DRST":0,"ODAY":"2019-10-
31","JRN":1490,"LINE":28,"START":"12:12","LOC":"GPS","STOP":null,"ROUTE":"1001","OCCU":0,
"SEQ":null}]},"errorMessage":null,"finalMessage":null,"terminal":false}
{"row":{"columns":[1572517399120,"/hfp/v2/journey/ongoing/vp/bus/0022/01115/4322/1/Vantaa
nkoski/11:40/4140232/5/60;24/28/81/18",{"DESI":"322","DIR":"1","OPER":22,"VEH":1115,"TST"
:"2019-10-
31T10:23:27.026Z","TSI":1572517407,"SPD":0.0,"HDG":25,"LAT":60.281248,"LONG":24.818611
,"ACC":-0.1,"DL":-13,"ODO":16929,"DRST":0,"ODAY":"2019-10-
31","JRN":77,"LINE":769,"START":"11:40","LOC":"GPS","STOP":"4140240","ROUTE":"4322","OCCU
":0,"SEQ":null}]},"errorMessage":null,"finalMessage":null,"terminal":false}
{"row":{"columns":[1572517399133,"/hfp/v2/journey/ongoing/vp/bus/0022/00746/4562/1/Aviapo
lis/12:04/4640208/5/60;25/20/88/20",{"DESI":"562","DIR":"1","OPER":22,"VEH":746,"TST":"20
19-10-
31T10:23:27.029Z","TSI":1572517407,"SPD":0.0,"HDG":280,"LAT":60.28208,"LONG":25.080237,"A
CC":0.0,"DL":51,"ODO":8384,"DRST":0,"ODAY":"2019-10-
31","JRN":126,"LINE":657,"START":"12:04","LOC":"GPS","STOP":null,"ROUTE":"4562","OCCU":0,
"SEQ":null}]},"errorMessage":null,"finalMessage":null,"terminal":false}
{"row":
null,"errorMessage":null,"finalMessage":"Limit Reached","terminal":true}
```


10. Let's try to map and filter the output a bit:

```
result = client.query('''
    select VP->oper, VP->veh, VP->lat, VP->long
    from vpositions
    where VP->oper=12 limit 10''')
for item in result: print(item)
```

In my case the output looked like this:

```
{"row":{"columns":[12,1309,60.228603,24.964448]},"errorMessage":null,"finalMessage":null,
"terminal":false}
{"row":{"columns":[12,1017,60.231048,24.89
2614]},"errorMessage":null,"finalMessage":null,"terminal":false}
{"row":{"columns":[12,1512,60.219296,25.133724]},"errorMessage":null,"finalMessage":null,
"terminal":false}
{"row":{"columns":[12,1610,60.184018,24.829178]},"errorMessage":null,"finalMessage":null,
"terminal":false}
{"row":{"columns":[12,921,60.17131,24.942813]},"errorMessage":null,"finalMessage":null,"t
erminal":false}
{"row":{"columns":[12,1210,60.193146,24.960069]},"errorMessage":null,"finalMessage":null,
"terminal":false}
{"row":{"columns":[12,1518,60.262631,24.886132]},"errorMessage":null,"finalMessage":null,
"terminal":false}
{"row":{"columns":[12,1524,60.244022,24.846748]},"errorMessage":null,"finalMessage":null,
"terminal":false}
{"row":{"columns":[12,1901,60.181659,24.927261]},"errorMessage":null,"finalMessage":null,
"terminal":false}
{"row":{"columns":[12,1330,60.186023,24.813034]},"errorMessage":null,"finalMessage":null,
"terminal":false}

{"row":null,"errorMessage":null,"finalMessage":"Limit Reached","terminal":true}
```


Evidently only the four fields `oper, veh, lat, long` are output, and the operator for all reported position records is `12`.

11. With this we have only scratched the surface of what is possible. Please refer to the link below for more information about the `ksql` library and its features. Try to do some more sophisticated operations using KSQL from within your Jupyter notebook.
https://github.com/bryanyang0528/ksql-python

Cleanup

1. Stop the Jupyter server with CTRL-C.
2. Stop the producer with this command:

```
$ docker container rm -f vp-producer
```

3. Execute the following command to completely cleanup your environment:

```
$ docker-compose down -v
```

Conclusion

In this exercise you have used KSQL from within a Jupyter notebook, to analyze data originating from the Kafka topic `vehicle-positions`. This topic has been populated by a producer you had created in an earlier lab. The producer consumed the data from the **High-frequency Positioning** service of **digitransit**.

STOP HERE. THIS IS THE END OF THE EXERCISE.

Appendix B: Running All Labs with Docker

Running Labs in Docker for Desktop

If you have installed Docker for Desktop on your Mac or Windows 10 Pro machine you are able to complete the course by building and running your applications from the command line.

- Increase the memory available to Docker Desktop to a minimum of 6 GiB. See the advanced settings for Docker Desktop for Mac, and Docker Desktop for Windows.
- Follow the instructions at → Preparing the Labs to `git clone` the source code, and launch the cluster containers with `docker-compose`. The exercise source code will now be on your host machine where you can edit the source code with any editor.
- Begin the exercises by first opening a bash shell on the tools container. All the command line instructions will work from the tools container. This container has been preconfigured with all of the tools you use in the exercises, e.g. `kafka-topics`, `gradle`, `dotnet` and `python`.

```
$ docker-compose exec tools /bin/bash
bash-4.4#
```

- Anywhere you are instructed to open additional terminal windows you can `exec` additional bash shells on the tools container with the same command as above on your host machine.
- Any subsequent `docker` or `docker-compose` instructions should be run on your host machine.

Running the Exercise Applications

From the `tools` container you can use command line alternatives to the VS Code steps used in the instructions. Complete the exercise code with an editor on your host machine, then use the following command line instructions to build and run the applications from the exercise directory.

- For Java applications: `gradle run`
- For C# applications: `dotnet run`
- For Python applications: `python3 main.py`

Where you are instructed to use **Debug** → **Stop Debugging** in VS code use CTRL-C to end the running exercise.

For example: to build and run the **Creating a Kafka Producer in Java** exercise. First make the source code updates to `~/confluent-dev/labs/create-producer-java/src/main/java/clients/Subscriber.java` on your host machine. Then enter the following into the bash shell on your tools container.

```
# If you haven't already, open a bash shell on the tools container
$ docker-compose exec tools /bin/bash
bash-4.4# ~/confluent-dev/labs/create-producer-java
bash-4.4# gradle run
```