Apache Kafka Fundamentals

Exercise Book

Version 7.1.1-v1.0.0

CONFLUENT

Table of Contents

Preamble

Copyright & Trademarks

Accessing your Lab Environment

1. Welcome to your lab environment! You are connected as user **training**, password **training**.
2. Open a new terminal window
3. Clone the **GitHub** repository with the sample solutions into the folder `~/confluent-fundamentals`:

   Bash:

   ```
   $ git clone --branch 7.1.1-v1.0.0 \
       https://github.com/confluentinc/training-fundamentals-src.git \
       ~/confluent-fundamentals
   ```

4. Navigate to the `~/confluent-fundamentals` folder:

   ```
   $ cd ~/confluent-fundamentals
   ```

5. Run a script to add entries to your `/etc/hosts` file. This allows you to refer to containers in your cluster via hostnames like `kafka` and `zookeeper`. If you are prompted for a password for the `training` account type `training` as the password:

   This step isn't needed if you are following the **Running Labs in Docker for Desktop** appendix.

   ```
   $ ~/confluent-fundamentals/update-hosts.sh
   Done!
   ```

   This applies to all subsequent command in this exercise book.

Docker Basics

This exercise book heavily relies on Docker (containers). Yet, to successfully complete all the exercises you do **NOT** need any previous knowledge of Docker. Though, if you have some basic knowledge of Docker it is definitely a plus.

If you do not have any prior knowledge of Docker and you have some free time then please try to familiarize yourself with the most fundamental concepts of Docker containers going through some of the interactive labs here: https://training.play-with-docker.com

STOP HERE. THIS IS THE END OF THE EXERCISE.

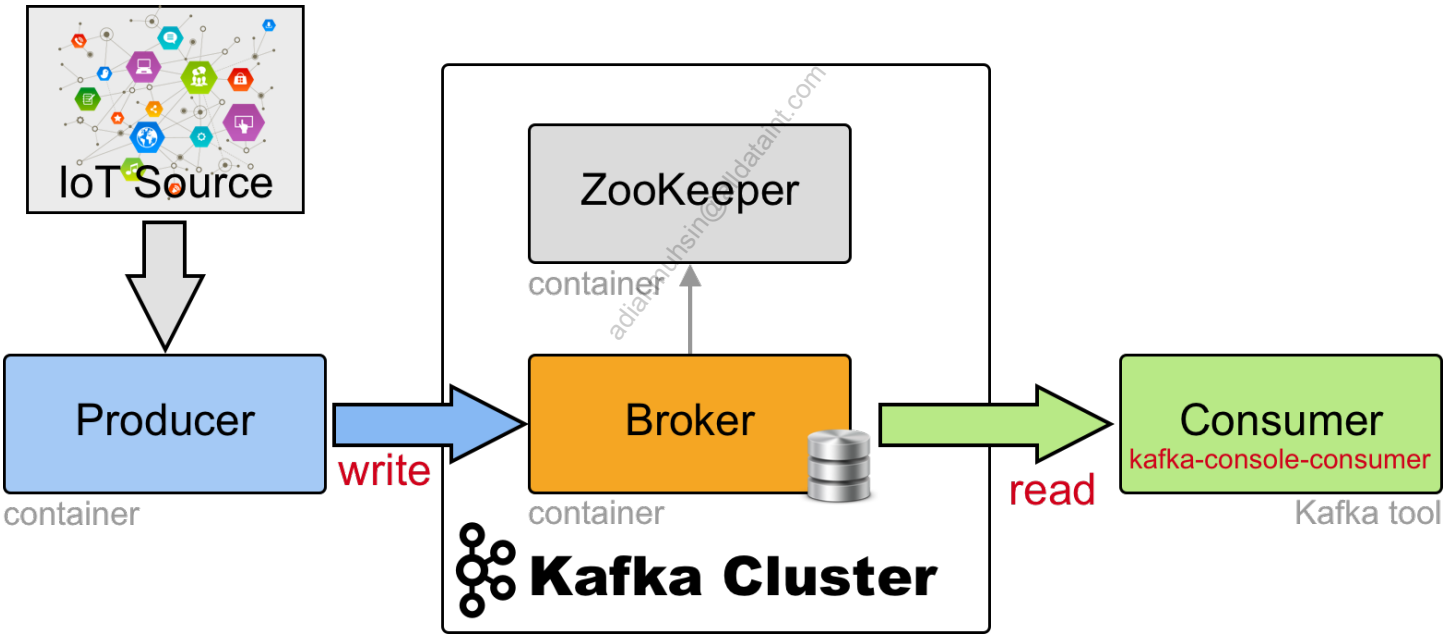
Lab 01 Exploring Apache Kafka

Exploring Apache Kafka

In this exercise we are going to explore a very simple Kafka cluster consisting of a single broker and a single ZooKeeper instance. We will also run a simple Kafka client - a producer - that writes some IoT data from a public source into Kafka. We will then use a Kafka tool called `kafka-console-consumer` to read that data from Kafka.

Don't worry if the terms **broker**, **ZooKeeper** and **Kafka client** don't make much sense to you, we will introduce them in detail in the next module.

Here is a sketch of how that looks:



For more info about our source of IoT data please refer to the following URL: https://digitransit.fi/en/developers/apis/4-realtime-api/vehicle-positions/

Running a simple Kafka Cluster

We are going to run all components of our simple Kafka cluster in Docker containers.

1. To do this navigate to the project folder and execute the `start.sh` script:

```
$ cd ~/confluent-fundamentals/labs/exploring
$ ./start.sh
```

Starting the cluster will take a moment, thus be patient. You will observe an output similar to this (shortened for readability):

```
...
Creating network "explore_confluent" with the default driver
Creating explore_producer_1  ... done
Creating explore_kafka_1     ... done
Creating explore_zookeeper_1 ... done
Waiting kafka to launch on 9092...
kafka not yet ready...
kafka not yet ready...
...
kafka is now ready!
Connection to kafka port 9092 [tcp/XmlIpcRegSvc] succeeded!
c5eaba41ac19d739d53e6e44b2909a36563ad22d428d5eac9cc5aaa2dbbea18b
```

2. Next use the tool `kafka-console-consumer` installed on your lab VM to read data that the producer is writing to Kafka. Use this command to do so:


```
$ kafka-console-consumer \
    --bootstrap-server kafka:9092 \
    --topic vehicle-positions
```

After a short moment you should see records being output on your terminal window in fast cadence. These records are live data coming from an MQTT source. Each record corresponds to a vehicle position (bus, tram or train) of the Finnish public transport provider. The data looks like this (shortened):

```
...
{"VP":{"desi":"65","dir":"1","oper":18,"veh":268,"tst":"2019-10-
18T08:38:31.978Z","tsi":1571387911,"spd":6.33,"hdg":353,"lat":60.2335
73,"long":24.978816,"acc":0.34,"dl":180,"odo":null,"drst":null,"oday"
:"2019-10-
18","jrn":127,"line":850,"start":"11:12","loc":"GPS","stop":null,"rou
te":"1065","occu":0}}
{"VP":{"desi":"643","dir":"1","oper":18,"veh":2983,"tst":"2019-10-
18T08:38:31.970Z","tsi":1571387911,"spd":12.98,"hdg":20,"lat":60.3740
53,"long":25.018964,"acc":-0.84,"dl":-
278,"odo":22802,"drst":null,"oday":"2019-10-
18","jrn":10,"line":1041,"start":"10:58","loc":"GPS","stop":null,"rou
te":"9643","occu":0}}
{"VP":{"desi":"633","dir":"2","oper":12,"veh":1812,"tst":"2019-10-
18T08:38:32.017Z","tsi":1571387912,"spd":0.00,"hdg":268,"lat":60.3231
16,"long":25.013476,"acc":0.00,"dl":-
120,"odo":6584,"drst":0,"oday":"2019-10-
18","jrn":44,"line":1074,"start":"11:25","loc":"GPS","stop":null,"rou
te":"4633","occu":0}}
{"VP":{"desi":"40","dir":"2","oper":12,"veh":825,"tst":"2019-10-
18T08:38:32.017Z","tsi":1571387912,"spd":0.09,"hdg":183,"lat":60.2069
46,"long":24.898891,"acc":-0.00,"dl":-
120,"odo":3927,"drst":1,"oday":"2019-10-
18","jrn":1467,"line":62,"start":"11:21","loc":"GPS","stop":1160109,"
route":"1040","occu":0}}
...
```


3. Stop the consumer by pressing **CTRL-c**.

MQTT stands for **Message Queuing Telemetry Transport**. It is a lightweight publish and subscribe system where you can publish and receive messages as a client. MQTT is a simple messaging protocol, designed for constrained devices with low-bandwidth. So, it's the perfect solution for **Internet of Things** (IoT) applications.

Cleanup

1. Before you end, please clean up your system by running the **stop.sh** script in the project folder:

```
$ ./stop.sh
```

You should see this:

```
producer
Stopping explore_zookeeper_1 ... done
Stopping explore_kafka_1     ... done
Removing explore_producer_1  ... done
Removing explore_zookeeper_1 ... done
Removing explore_kafka_1     ... done
Removing network explore_confluent
```

Conclusion

In this exercise we created a simple real-time data pipeline powered by Kafka. We have written data that originates from a public IoT data source into a simple Kafka cluster. This data we then have consumed with a simple Kafka tool called `kafka-console-consumer`.

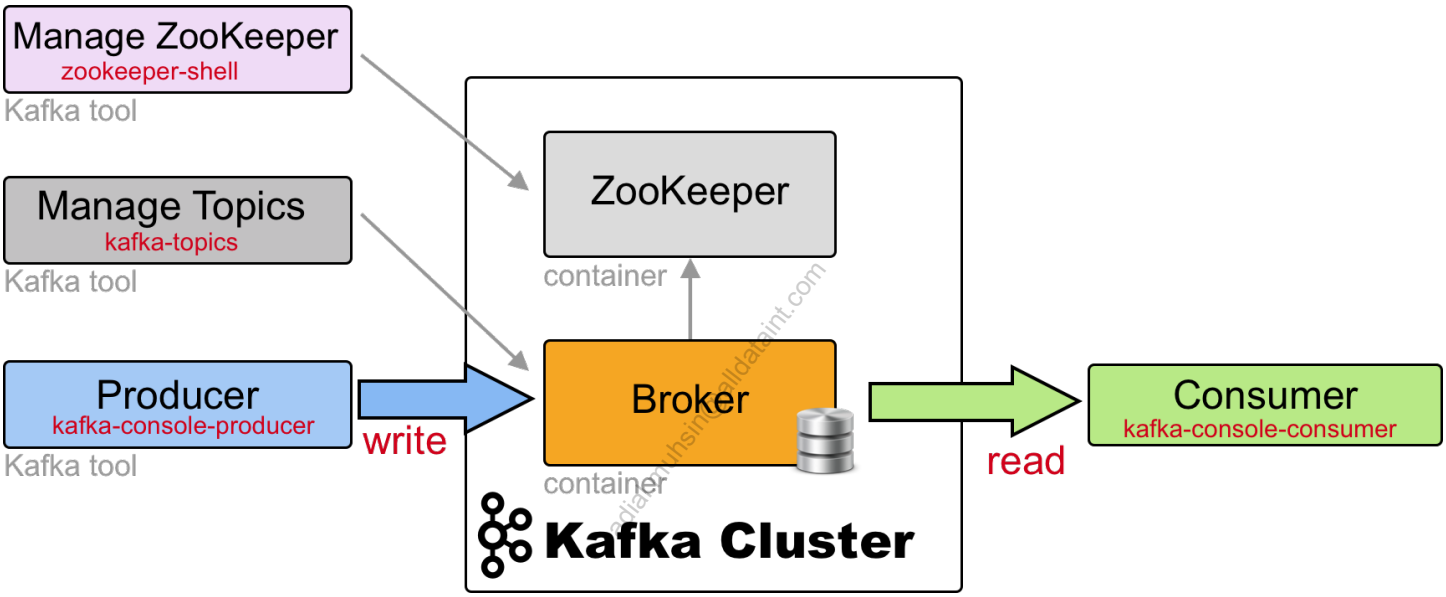
STOP HERE. THIS IS THE END OF THE EXERCISE.

Lab 02 Fundamentals of Apache Kafka

Fundamentals of Apache Kafka

In this exercise we are going to use a basic Kafka cluster consisting of a single broker and ZooKeeper instance. Once again we are going to run all components of our Kafka cluster in Docker containers.

Here is a sketch of the lab setup:



Prerequisites

1. Run the Kafka cluster by navigating to the project folder and executing the **`start.sh`** script:

```
$ cd ~/confluent-fundamentals/labs/fundamentals
$ ./start.sh
```

Working with Topics

1. Let's first use the **`kafka-topics`** tool to list all topics registered on the Kafka cluster:

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

you should see this:

```
_confluent-metrics
```

Apparently there is only a single topic called `_confluent-metrics` registered in the cluster.

2. Now let's create a topic called `vehicle-positions` with 6 partitions and a `replication-factor` of `1`:

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

3. To verify the details of the topic just created we can use the `--describe` parameter:

```
$ kafka-topics --bootstrap-server kafka:9092 \
    --describe \
    --topic vehicle-positions
```

giving us this:

```
Topic: vehicle-positions    TopicId: f5HGgfhBTliEK7zLkBgZxg
PartitionCount: 6   ReplicationFactor: 1    Configs:
segment.bytes=536870912,retention.bytes=536870912
    Topic: vehicle-positions    Partition: 0    Leader: 101 Replicas:
101   Isr: 101    Offline:
    Topic: vehicle-positions    Partition: 1    Leader: 101 Replicas:
101   Isr: 101    Offline:
    Topic: vehicle-positions    Partition: 2    Leader: 101 Replicas:
101   Isr: 101    Offline:
    Topic: vehicle-positions    Partition: 3    Leader: 101 Replicas:
101   Isr: 101    Offline:
    Topic: vehicle-positions    Partition: 4    Leader: 101 Replicas:
101   Isr: 101    Offline:
    Topic: vehicle-positions    Partition: 5    Leader: 101 Replicas:
101   Isr: 101    Offline:
```


We can see a line for each partition created. For each partition we get the leader broker, the replica placement and the ISR list (In-sync Replica list). In our case the list looks simple since we only have one single replica, placed on broker `101`.

4. Try to create another topic called `test-topic` with 3 partitions and replication factor 1.

5. List all the topics in your Kafka cluster. You should see this:

```
_confluent-metrics
test-topic
vehicle-positions
```

6. To delete the topic `test-topic` use the following command:

```
$ kafka-topics --bootstrap-server kafka:9092 \
    --delete \
    --topic test-topic
```

7. Double check that the topic is gone by listing all topics in the cluster.

Producing and consuming data

Let's add some data to a topic `sample-topic` by running the command line tool `kafka-console-producer`. Then we will use the tool `kafka-console-consumer` to consume the same data.

1. Use the tool `kafka-topics` to create a topic called `sample-topic` with 3 partitions and a replication factor of 1.

If you forgot how to do this, then have a look at how we created the topic `vehicle-positions` earlier in this exercise.

2. Run the `kafka-console-producer` command line tool:

```
$ kafka-console-producer --broker-list kafka:9092 \
    --topic sample-topic
```

3. Type `hello` at the prompt and hit `<Enter>`:

```
>hello
```

4. Type a few more lines at the prompt (each terminated with `<Enter>`):

```
>world
>Kafka
>is
>cool!
```

and press `CTRL-d` when done to exit the producer.

5. Now use the `kafka-console-consumer` to consume all data **from the beginning** of topic `sample-topic`:

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

In my case the output looks like this:

```
hello
Kafka
is
world
cool!
```

Notice how the order of the items entered is scrambled. Please take a moment to reflect why this happens. Discuss your findings with your peers.

6. End the consumer by pressing `CTRL-c`.

7. So far we have produced and consumed data without a key. Let's now run the producer in a way that we can also enter a key with each value:

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

The last two parameters tell the producer to expect a key and to use the comma (,) as a separator between key and value at the input.

8. Enter some values at the prompt:

```
>1,apples
>2,pears
>3,walnuts
>4,peanuts
>5,oranges
```

and press `CTRL-d` to exit the producer.

9. Let's use the console consumer tool and configure it such as that it outputs keys and values:

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

the output in my case looks like this:

```
null    hello
null    Kafka
1       apples
5       oranges
null    is
4       peanuts
null    world
null    cool!
2       pears
3       walnuts
```

Notice how `null` is output for the key for the values we first entered without defining a key.

10. Once again the question is: "Why is the order of the items read from the topic not the order that you produced the messages?"
11. Press `CTRL-c` to exit the consumer.

Using the ZooKeeper Shell

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

```
$ zookeeper-shell zookeeper
```

which will output the following in your terminal:

```
Connecting to zookeeper
Welcome to ZooKeeper!
JLine support is disabled

WATCHER::

WatchedEvent state:SyncConnected type:None path:null
```

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

```
ls /
```

   In our case you should get something like this:

```
[admin, brokers, cluster, config, consumers, controller,
controller_epoch, feature, isr_change_notification,
latest_producer_id_block, leadership_priority,
log_dir_event_notification, zookeeper]
```


3. Let's see what the `brokers` node reveals:

```
ls /brokers

[ids, seqid, topics]
```

4. And what do we find under the node `ids`?

```
ls /brokers/ids

[101]
```

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

5. Try to find out what's to be found in other nodes of the ZooKeeper data tree, e.g. to find out something about the cluster itself use:

```
get /cluster/id

{"version":"1","id":"Rslk7ZJnRsGFfeHwfwhzmw"}
```

Here I can see that my cluster ID is equal to `Rslk7ZJnRsGFfeHwfwhzmw`.

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

Cleanup

1. Before you end, please clean up your system by running the `stop.sh` script in the project folder:

```
$ ./stop.sh
```

Conclusion

In this exercise we created a simple Kafka cluster. We then used the tool `kafka-topics` to list, create, describe and delete topics. Next we used the tools `kafka-console-producer` and `kafka-console-consumer` to produce data to and consume data from Kafka. Finally we used the ZooKeeper shell to analyze some of the data stored by Kafka in ZooKeeper.

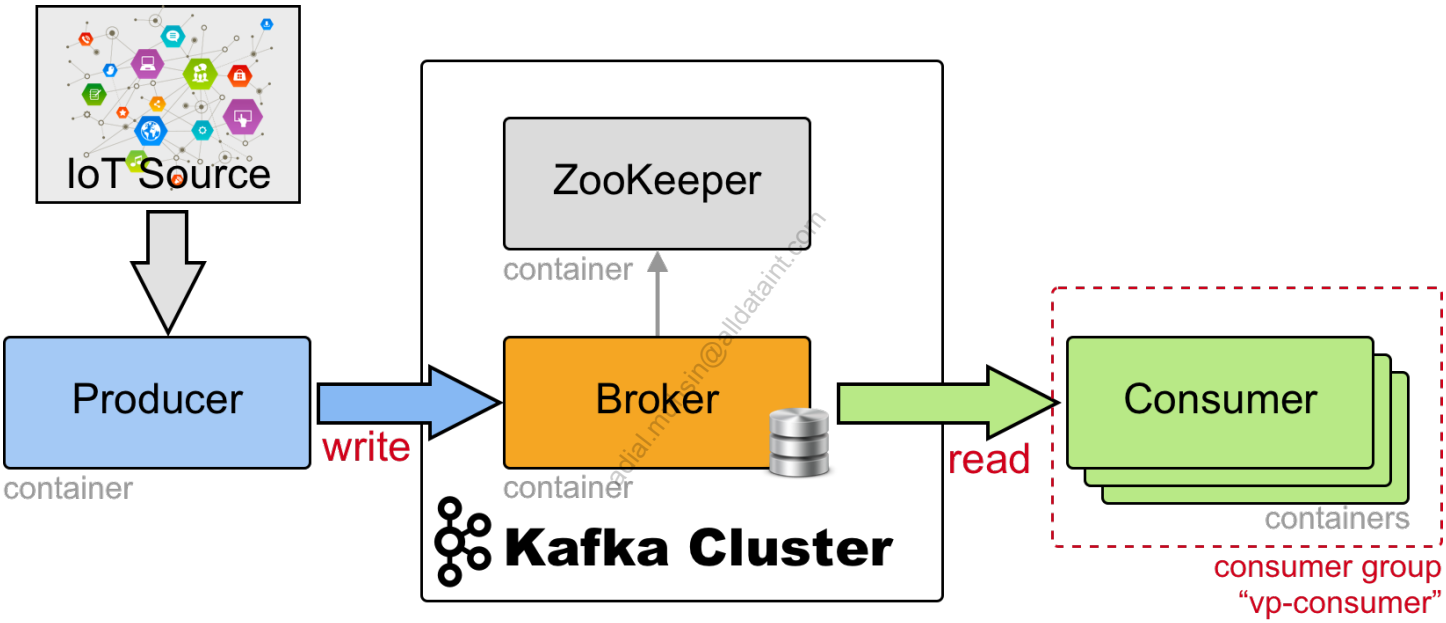
STOP HERE. THIS IS THE END OF THE EXERCISE.

Lab 03 How Kafka Works

How Kafka Works

In this lab we are going to create a simple Java consumer. The consumer will read data from the topic `vehicle-positions` that the producer we introduced in an earlier lab produces from our trusted IoT source. In a second step we will scale the consumer (or consumer group) and analyze the effect.

Here is a simple graphic of the lab setup:



Prerequisites

1. Run the Kafka cluster by navigating to the project folder and executing the `start.sh` script:

```
$ cd ~/confluent-fundamentals/labs/advanced-topics
$ ./start.sh
```

This will start our mini Kafka cluster, create the topic `vehicle-positions`, and then start the producer that is writing data from our IoT source to the topic.

Investigating the Consumer

1. Navigate to the **consumer** subfolder of the project folder:

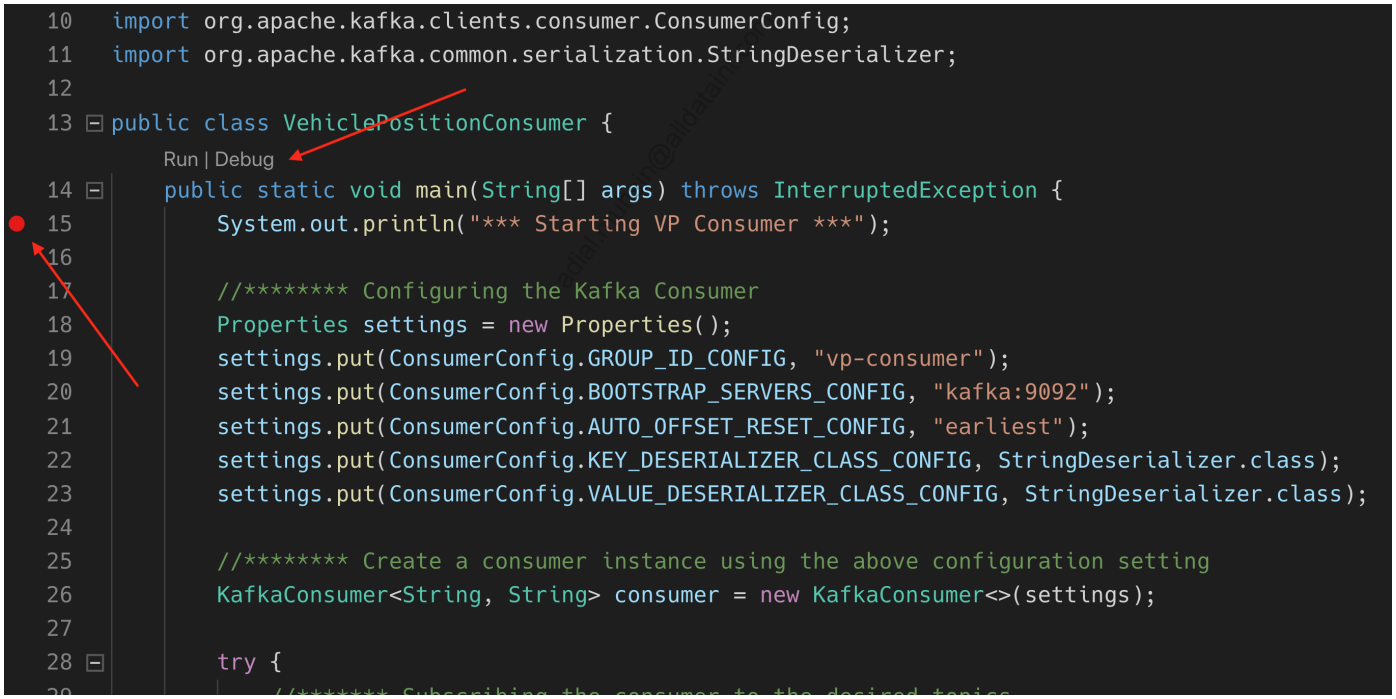
```
$ cd ~/confluent-fundamentals/labs/advanced-topics/consumer
```

2. Start Visual Studio Code from within this folder:

```
$ code .
```

3. Locate the file **VehiclePositionConsumer.java** at **src/main/java/clients** and open it. Try to analyze what the code does.

4. To set a breakpoint at line 15, click the left margin and then click on the **Debug** link right above line 14 to start debugging (or use the menu **Run→Start Debugging**):



The application should start and code execution should stop at line 15:

```
12  public class VehiclePositionConsumer {
        Run | Debug
13      public static void main(String[] args) {
14          System.out.println("*** Starting VP Consumer ***");
15
16          //******** Configuring the Kafka Consumer
17          Properties settings = new Properties();
18          settings.put(ConsumerConfig.GROUP_ID_CONFIG, "vp-consumer");
```

5. Use the various buttons on the debugging toolbar to navigate through the code either step by step or continue clicking the continue button:

6. If you let the app run your output in the **DEBUG CONSOLE** should look similar to this:

```
* Starting VP Consumer *
...
offset = 4728, key =
/hfp/v1/journey/ongoing/bus/0012/01806/2551/1/Westendinas./14:11/2213
206/4/60;24/18/82/13, value = {"VP":{"long":24.823896,"oday":"2019-
06-
21","lat":60.181576,"odo":14370,"oper":12,"desi":"551","veh":1806,"ts
t":"2019-06-
21T11:46:03Z","dir":"1","tsi":1561117563,"hdg":207,"start":"14:11","d
l":34,"jrn":86,"line":842,"spd":0.48,"drst":0,"acc":0.39}}
offset = 4729, key =
/hfp/v1/journey/ongoing/bus/0022/00625/4624/2/Tikkurila/14:37/4700210
/4/60;25/30/26/22, value = {"VP":{"long":25.062562,"oday":"2019-06-
21","lat":60.322748,"odo":3694,"oper":22,"desi":"624","veh":625,"tst"
:"2019-06-
21T11:46:03Z","dir":"2","tsi":1561117563,"hdg":236,"start":"14:37","d
l":-60,"jrn":152,"line":809,"spd":3.62,"drst":0,"acc":0.23}}
...
```

7. Stop the application with the **Stop** button on the debug toolbar.
8. Exit VS Code when done.

Building the Consumer Docker image

To prepare ourselves to scale the consumer up and down we want to run each consumer instance as a Docker container. We have a Dockerfile for this in our project folder.

1. Navigate to the `consumer` subfolder of the project folder:

```
$ cd ~/confluent-fundamentals/labs/advanced-topics/consumer
```

2. To build the Docker image run the script `build-consumer.sh`:

```
$ ./build-image.sh
```

please be patient, this takes a moment or two. You should see something like this in your terminal (shortened for readability):

```
Sending build context to Docker daemon  33.28kB
Step 1/13 : FROM gradle:5.2.1-jdk11-slim AS builder
 ---> 22bd673b8ca6
Step 2/13 : WORKDIR /home/gradle/project
 ---> Using cache
 ---> f329118c9919
Step 3/13 : COPY build.gradle ./
 ---> Using cache
 ---> a8f70d1286d9
...
Step 12/13 : WORKDIR /app/project
 ---> Running in dc8f7895f854
Removing intermediate container dc8f7895f854
 ---> d6a3e36f38d3
Step 13/13 : CMD java -classpath "lib/*"
clients.VehiclePositionConsumer
 ---> Running in c842fd617a64
Removing intermediate container c842fd617a64
 ---> 04963927cc8b
Successfully built 04963927cc8b
Successfully tagged sample-consumer:1.0
```


Running and scaling our consumer

1. Run your first consumer (an instance of the Docker image you just built) as follows:

```
$ ./run-consumer.sh
```

you should see a long ID, something like this:

```
97d34f47f2e477f66f360cd6b0e83bb...
```

indicating that a container with the above ID is running your consumer (in the background, as a daemon).

2. Let's use the `kafka-consumer-groups` tool to see what's going on:

```
$ kafka-consumer-groups \
  --bootstrap-server kafka:9092 \
  --group vp-consumer \
  --describe
```

we get this kind of output:

```
GROUP           TOPIC             PARTITION  CURRENT-OFFSET  LOG-END-
OFFSET  LAG             CONSUMER-ID
HOST            CLIENT-ID
vp-consumer     vehicle-positions 2          0               6607
6607            consumer-vp-consumer-1-0b9d07d9-322b-46c7-a11c-
b822aabab902 /172.21.0.5     consumer-vp-consumer-1
vp-consumer     vehicle-positions 3          147             7151
7004            consumer-vp-consumer-1-0b9d07d9-322b-46c7-a11c-
b822aabab902 /172.21.0.5     consumer-vp-consumer-1
vp-consumer     vehicle-positions 1          0               7689
7689            consumer-vp-consumer-1-0b9d07d9-322b-46c7-a11c-
b822aabab902 /172.21.0.5     consumer-vp-consumer-1
vp-consumer     vehicle-positions 4          2353            6183
3830            consumer-vp-consumer-1-0b9d07d9-322b-46c7-a11c-
b822aabab902 /172.21.0.5     consumer-vp-consumer-1
vp-consumer     vehicle-positions 5          1500            7443
5943            consumer-vp-consumer-1-0b9d07d9-322b-46c7-a11c-
b822aabab902 /172.21.0.5     consumer-vp-consumer-1
vp-consumer     vehicle-positions 0          0               6881
6881            consumer-vp-consumer-1-0b9d07d9-322b-46c7-a11c-
b822aabab902 /172.21.0.5     consumer-vp-consumer-1
```

We can see that we have a single client (`consumer-vp-consumer-1-0b9…`) consuming from all 6 partitions. We can also see the current lag on each partition. The latter being the so called **consumer lag**, which is an indication whether or not the consumer group is falling behind.

3. Repeat the above command a few times and observe how the **LAG** behaves.

You may want to use the **watch** command to observe how the **LAG** is varying over time:

```
$ watch kafka-consumer-groups \
    --bootstrap-server kafka:9092 \
    --group vp-consumer \
    --describe
```

End watching by pressing **CTRL-c**.

4. Let's run another consumer and scale up to 2 instances:

```
$ ./run-consumer.sh
```

5. And describe the consumer group **vp-consumer** again:

```
$ kafka-consumer-groups --bootstrap-server kafka:9092 \
  --group vp-consumer \
  --describe

GROUP           TOPIC                PARTITION  CURRENT-OFFSET  LOG-END-
OFFSET  LAG             CONSUMER-ID                     HOST
CLIENT-ID
vp-consumer     vehicle-positions 0          14878           14972
94              consumer-vp-consumer-1-0b9... /172.21.0.5
consumer-vp-consumer-1
vp-consumer     vehicle-positions 1          16895           16971
76              consumer-vp-consumer-1-0b9... /172.21.0.5
consumer-vp-consumer-1
vp-consumer     vehicle-positions 2          15186           15278
92              consumer-vp-consumer-1-0b9... /172.21.0.5
consumer-vp-consumer-1
vp-consumer     vehicle-positions 3          16271           16330
59              consumer-vp-consumer-1-84f... /172.21.0.6
consumer-vp-consumer-1
vp-consumer     vehicle-positions 4          14009           14066
57              consumer-vp-consumer-1-84f... /172.21.0.6
consumer-vp-consumer-1
vp-consumer     vehicle-positions 5          17284           17339
55              consumer-vp-consumer-1-84f... /172.21.0.6
consumer-vp-consumer-1
```


Now we see, that there are 2 consumer instances **consumer-vp-consumer-1-0b9…** and **consumer-vp-consumer-1-84f…** sharing their work. A fully transparent rebalance of the

workload has happened!
Please also note that the **LAG** does not grow so fast as before since we have parallelized the workload.

6. Scale up the consumer group further and observe how the consumer lag behaves.
7. What happens if you scale the consumer group to more than 6 instances?

Cleanup

1. Before you end, please clean up your system by running the `stop.sh` script in the project folder:

```
$ cd ~/confluent-fundamentals/labs/advanced-topics
$ ./stop.sh
```

Conclusion

In this lab we have built and run a simple Kafka consumer. We then have scaled the consumer up and analyzed the effect of the scaling using the tool `kafka-consumer-groups`.

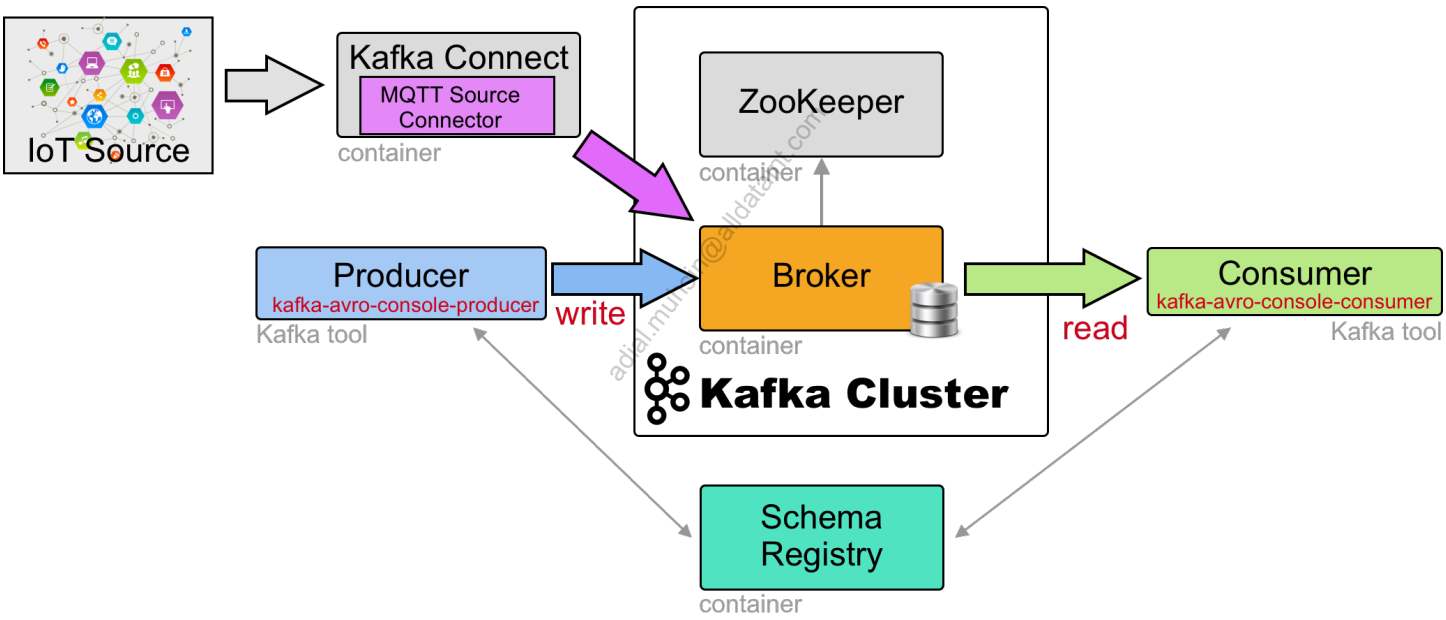
STOP HERE. THIS IS THE END OF THE EXERCISE.

Lab 04 Integrating Kafka into your Environment

Integrating Kafka into your Environment

In this lab we are going to use the Confluent Schema Registry to manage the schemas of the messages we'll be generating, where their values will be formatted in Avro. You will also be using an MQTT source connector in Kafka Connect to import data from an IoT data source.

Here is a simple graphic of the lab setup:



Prerequisites

1. Run a simple Kafka cluster, including Confluent Schema Registry, by navigating to the project folder and executing the `start.sh` script:

```
$ cd ~/confluent-fundamentals/labs/ecosystem
$ ./start.sh
```

2. Let's create the topic called `stations-avro` we need for our sample:

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

Generating and Consuming Avro Messages

1. Define an Avro schema for our records and assign it to a variable `SCHEMA`. The command to do so looks like this:

```
$ export SCHEMA='{
    "type":"record",
    "name":"station",
    "fields":[
        {"name":"city","type":"string"},
        {"name":"country","type":"string"}
    ]
}'
```


2. Now let's use this schema and provide it as an argument to the `kafka-avro-console-producer` tool:

```
$ kafka-avro-console-producer \
  --broker-list kafka:9092 \
  --topic stations-avro \
  --property schema.registry.url=http://schema-registry:8081 \
  --property value.schema="$SCHEMA"
```

We pass the information about the location of the Schema Registry and the schema itself as properties to the tool.

3. Now let's add some data. Enter the following values to the producer:

```
{"city": "Pretoria", "country": "South Africa"}
{"city": "Cairo", "country": "Egypt"}
{"city": "Nairobi", "country": "Kenya"}
{"city": "Addis Ababa", "country": "Ethiopia"}
```

4. Exit the `kafka-avro-console-producer` by pressing `CTRL-c`.

5. Run the `kafka-avro-console-consumer` to read the Avro formatted messages:

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

   And you should see this output:

```
{"city":"Pretoria","country":"South Africa"}
{"city":"Cairo","country":"Egypt"}
{"city":"Nairobi","country":"Kenya"}
{"city":"Addis Ababa","country":"Ethiopia"}
```

6. Stop the Avro consumer by pressing `CTRL-c`.

Using Kafka Connect to Import MQTT Data

In this exercise we will use the MQTT source connector from Confluent Hub (https://confluent.io/hub) to import data from our trusted MQTT source.

1. Create a topic `vehicle-positions` with 6 partitions and replication factor 1.

   If you forgot how to do this, then have a look at how we created the topic `vehicle-positions` in the exercise **Fundamentals of Apache Kafka**. Alternatively simply type `kafka-topics` at the command line and hit `<Enter>`. A description of all options will be output.

2. Kafka Connect is running as part of our cluster. It already has the MQTT Connector installed from Confluent Hub. Create a 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/train/#",
        "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 .
```

This will import position data of all **trains**.

If you are curious, you can find the Dockerfile in the subfolder `connect` of the project folder, which we use to install the requested MQTT connector from Confluent Hub on top of the official Confluent Connector image.

3. 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**.

4. 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":"K","dir":"2","oper":90,"veh":1036,"tst":"2019-10-
18T09:10:27.021Z","tsi":1571389827,"spd":0.20,"hdg":199,"lat":60.3326
60,"long":25.068842,"acc":0.07,"dl":-
15,"odo":8187,"drst":1,"oday":"2019-10-
18","jrn":9304,"line":280,"start":"12:02","loc":"GPS","stop":4730551,
"route":"3001K","occu":0}}
{"VP":{"desi":"P","dir":"1","oper":90,"veh":1035,"tst":"2019-10-
18T09:10:27.046Z","tsi":1571389827,"spd":0.00,"hdg":24,"lat":60.23028
9,"long":24.883413,"acc":0.00,"dl":-
60,"odo":7567,"drst":0,"oday":"2019-10-
18","jrn":8657,"line":636,"start":"11:58","loc":"GPS","stop":1294552,
"route":"3002P","occu":0}}
{"VP":{"desi":"U","dir":"1","oper":90,"veh":1075,"tst":"2019-10-
18T09:10:27.148Z","tsi":1571389827,"spd":0.00,"hdg":177,"lat":60.1734
87,"long":24.940247,"acc":0.00,"dl":120,"odo":0,"drst":1,"oday":"2019
-10
-18","jrn":8463,"line":292,"start":"12:12","loc":"GPS","stop":1020502
,"route":"3002U","occu":0}}
{"VP":{"desi":"K","dir":"2","oper":90,"veh":1064,"tst":"2019-10-
18T09:10:27.153Z","tsi":1571389827,"spd":0.00,"hdg":9,"lat":60.403977
,"long":25.106410,"acc":0.00,"dl":120,"odo":8,"drst":1,"oday":"2019-
10-
18","jrn":9308,"line":280,"start":"12:12","loc":"GPS","stop":9040502,
"route":"3001K","occu":0}}
{"VP":{"desi":"I","dir":"1","oper":90,"veh":1034,"tst":"2019-10-
18T09:10:27.163Z","tsi":1571389827,"spd":0.00,"hdg":174,"lat":60.2613
66,"long":24.854879,"acc":0.00,"dl":0,"odo":37563,"drst":0,"oday":"20
19-10-
18","jrn":9048,"line":279,"start":"11:26","loc":"GPS","stop":4150501,
"route":"3001I","occu":0}}
Processed a total of 5 messages
```


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

Cleanup

1. Before you end, please clean up your system by running the **`stop.sh`** script in the project folder:

```
$ ./stop.sh
```

Conclusion

In this lab we have defined an Avro schema used to serialize and deserialize the value part of

our messages that we write to and the read from a topic in Kafka. We used the two command line tools `kafka-avro-console-producer` and `kafka-avro-console-consumer` for this task. We also used an MQTT source connector in Kafka Connect to import data from an IoT data source.

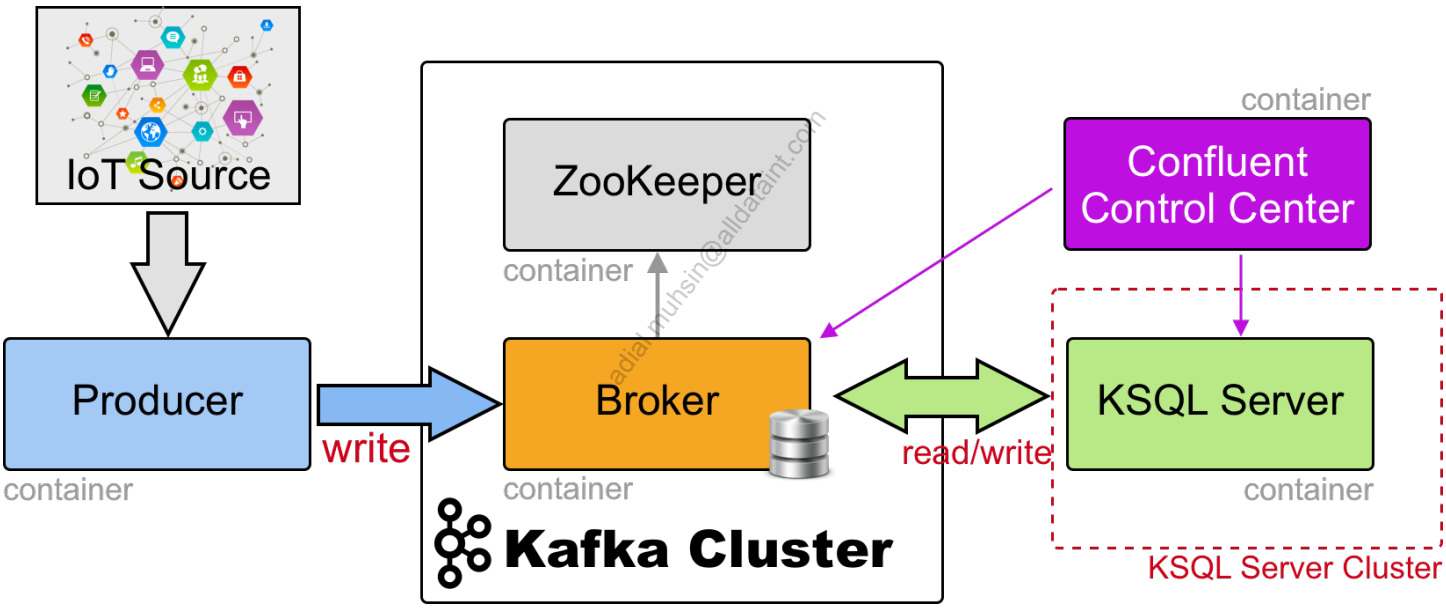
STOP HERE. THIS IS THE END OF THE EXERCISE.

Lab 05 The Confluent Platform

The Confluent Platform

In this exercise we are going to use 3 parts of the Confluent Platform that add tremendous value to a real-time stream processing platform powered by Apache Kafka®. We will use Confluent Control Center to monitor and analyze our Kafka cluster, and to inspect our topic `vehicle-positions`. We will also leverage the ksqlDB integration of Control Center to get our feet wet with ksqlDB. Finally we will use the ksqlDB CLI to do some simple stream processing.

Here is a simple graphic of the lab setup:



Prerequisites

1. Run the application by navigating to the project folder and executing the `start.sh` script:

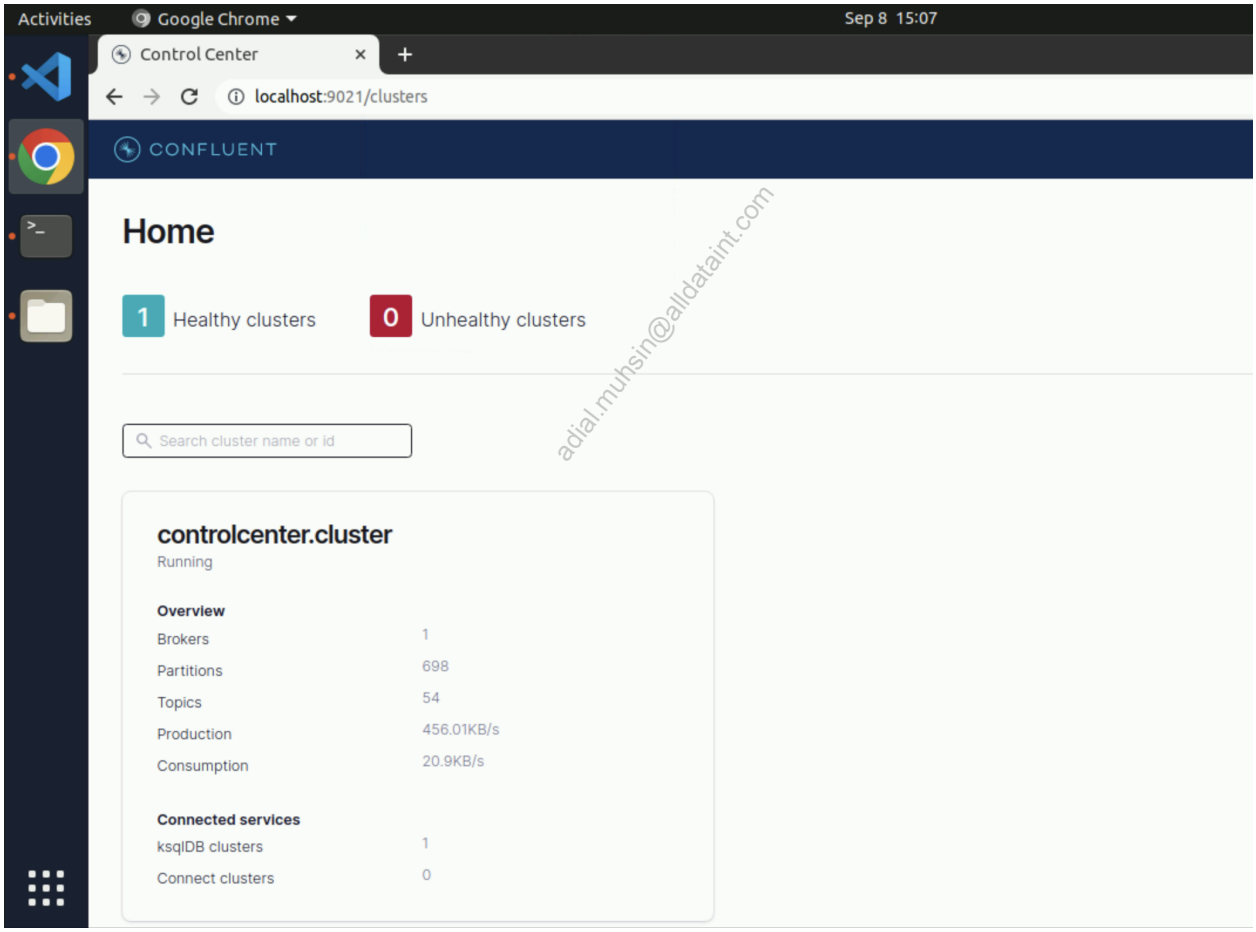
```
$ cd ~/confluent-fundamentals/labs/confluent-platform
$ ./start.sh
```

This will start the Kafka cluster, ksqlDB Server, Confluent Control Center and the producer. It will take approximately 2 minutes for Control Center to start serving.

Under certain circumstances the producer may fail and no (more) data is generated. If this happens to you, then the best solution is to start over. Execute `./stop.sh` on the command line, and then execute `./start.sh` again.

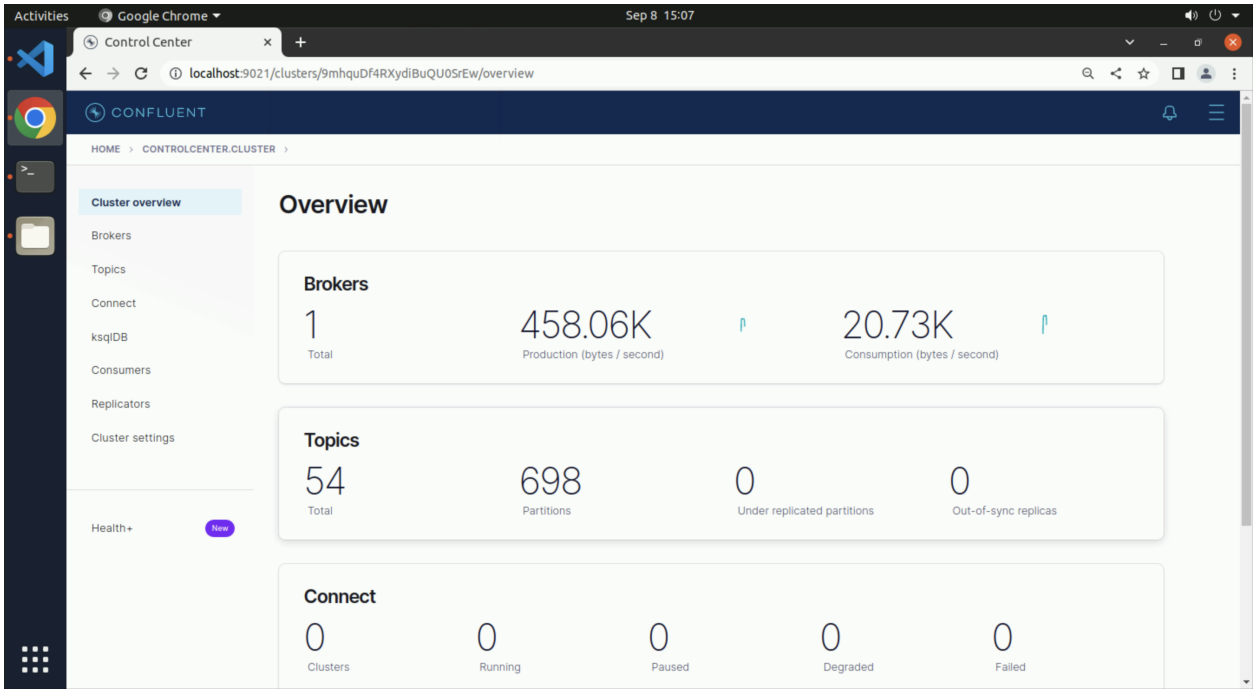
Monitoring and Inspecting our Kafka cluster with Confluent Control Center

1. Open a browser window and navigate to http://localhost:9021. You should see this:



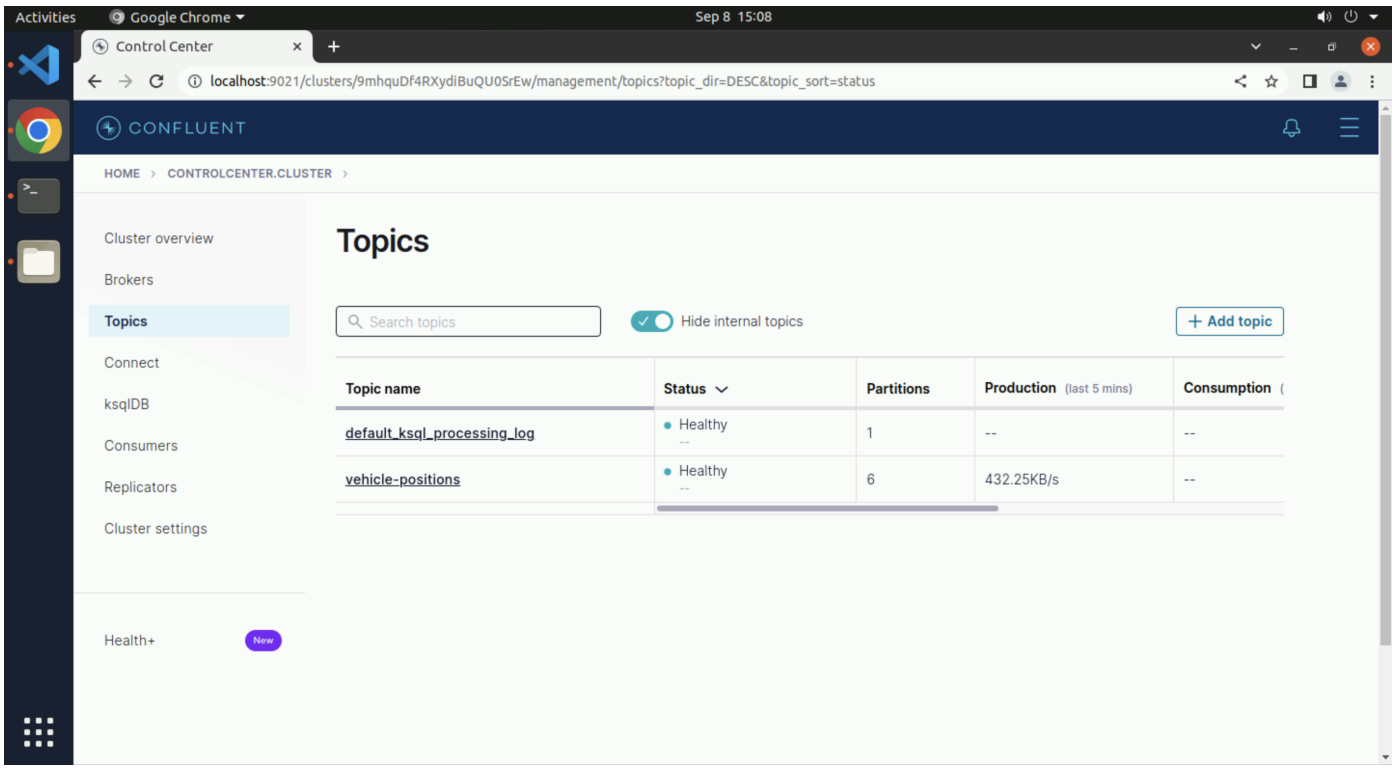
If your cluster is shown as **unhealthy** then you may have to wait a few moments until the cluster has stabilized.

2. Select **controlcenter.cluster** and you'll see a view called **Cluster overview** showing a set of cluster metrics that are indicators for the overall health of the Kafka cluster:



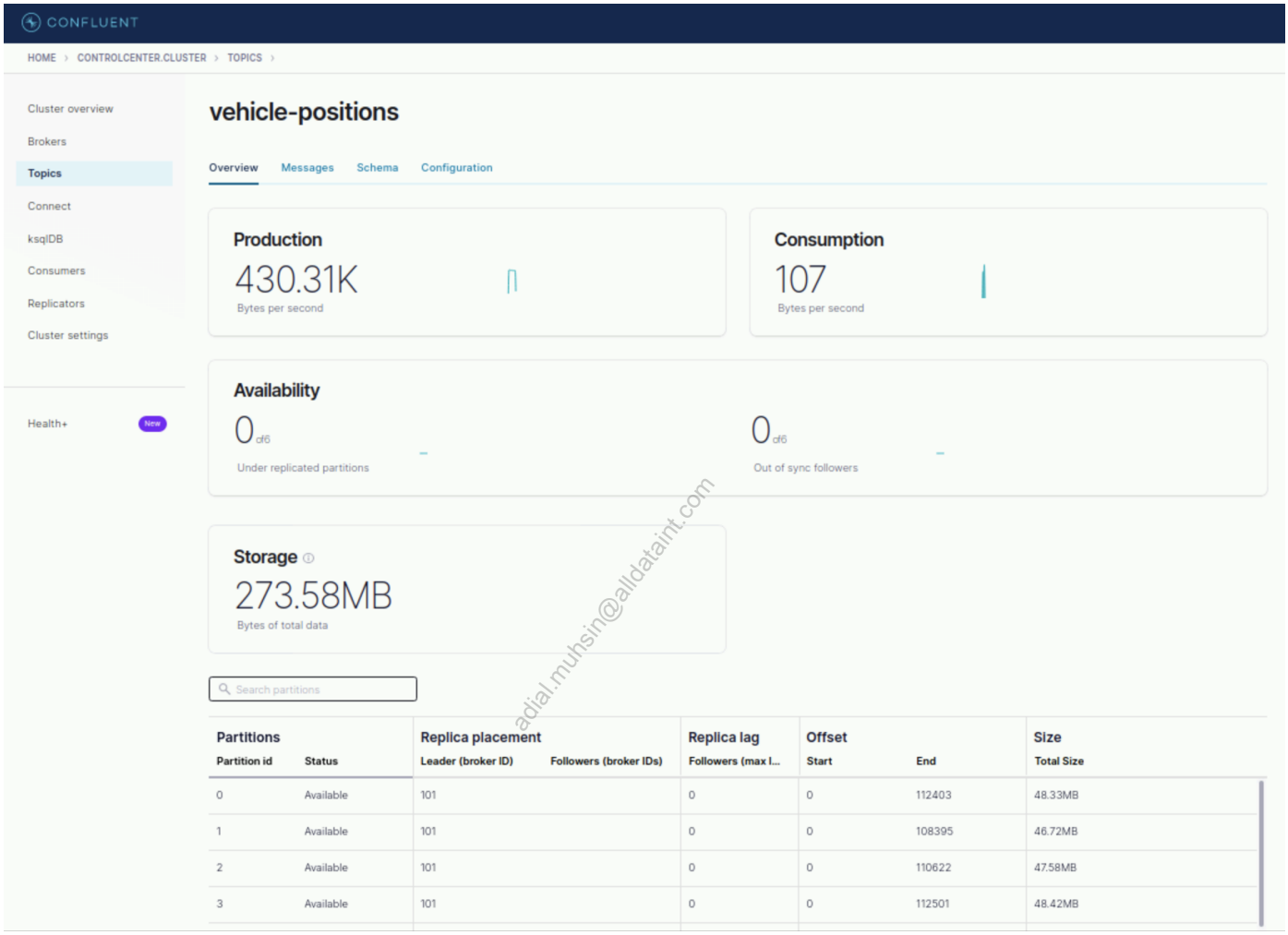
Notice the **Control Center Cluster** tabs on the left with the items **`Brokers`**, **`Topics`**, **`Connect`** and so on. We are currently on the **`Cluster overview`** tab.

3. From the list of tabs on the left select **Topics**. You should see this:



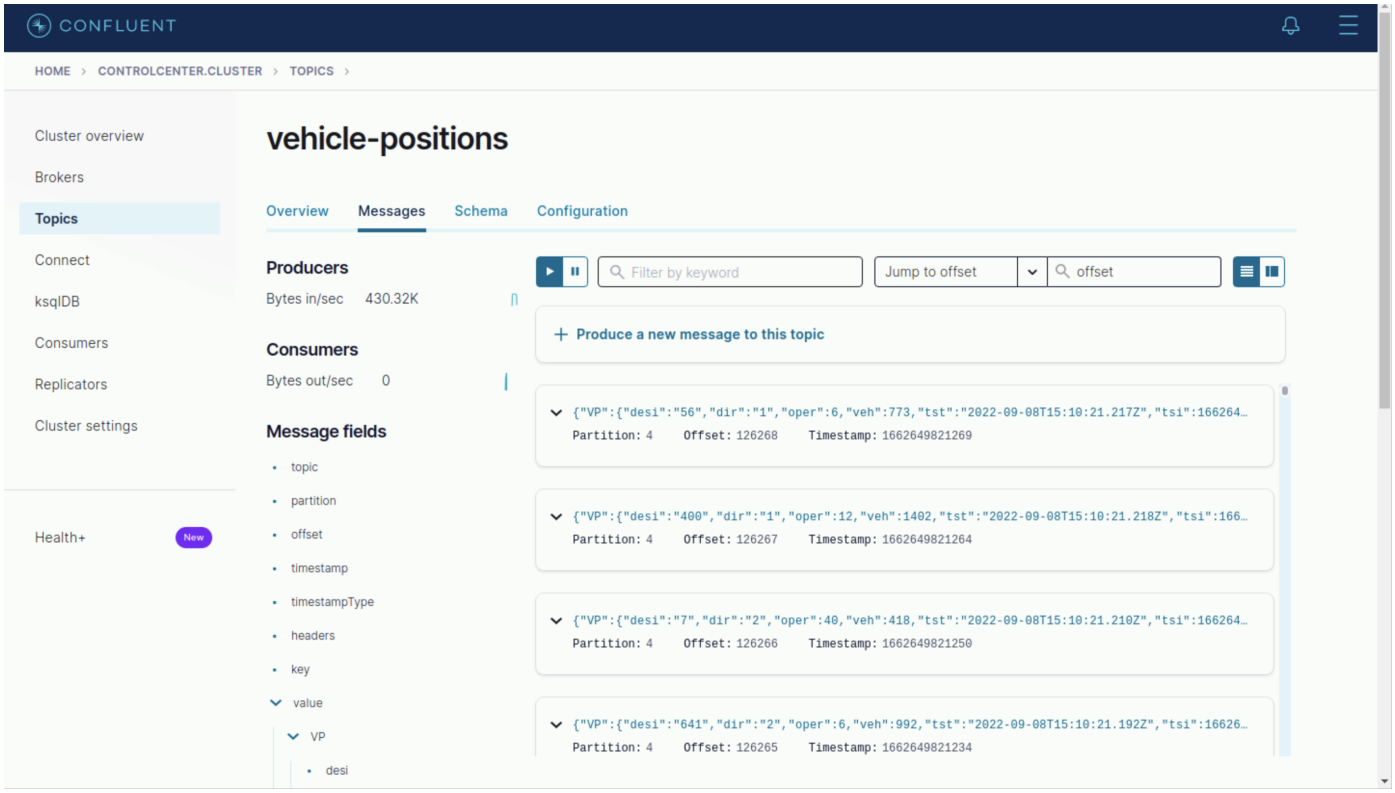
We currently have two topics in our cluster `vehicle-positions` and `default_ksql_processing_log` (created automatically by ksqlDB for its internal use).

4. Click on the topic `vehicle-positions` and you will be transferred to the topic **Overview** page:



Here you see that our topic has 6 partitions and that they all reside on broker `101` (the only one we have). Notice that this is a tabbed view and we are on the **Overview** tab.

5. Click on the **Messages** tab to see this:

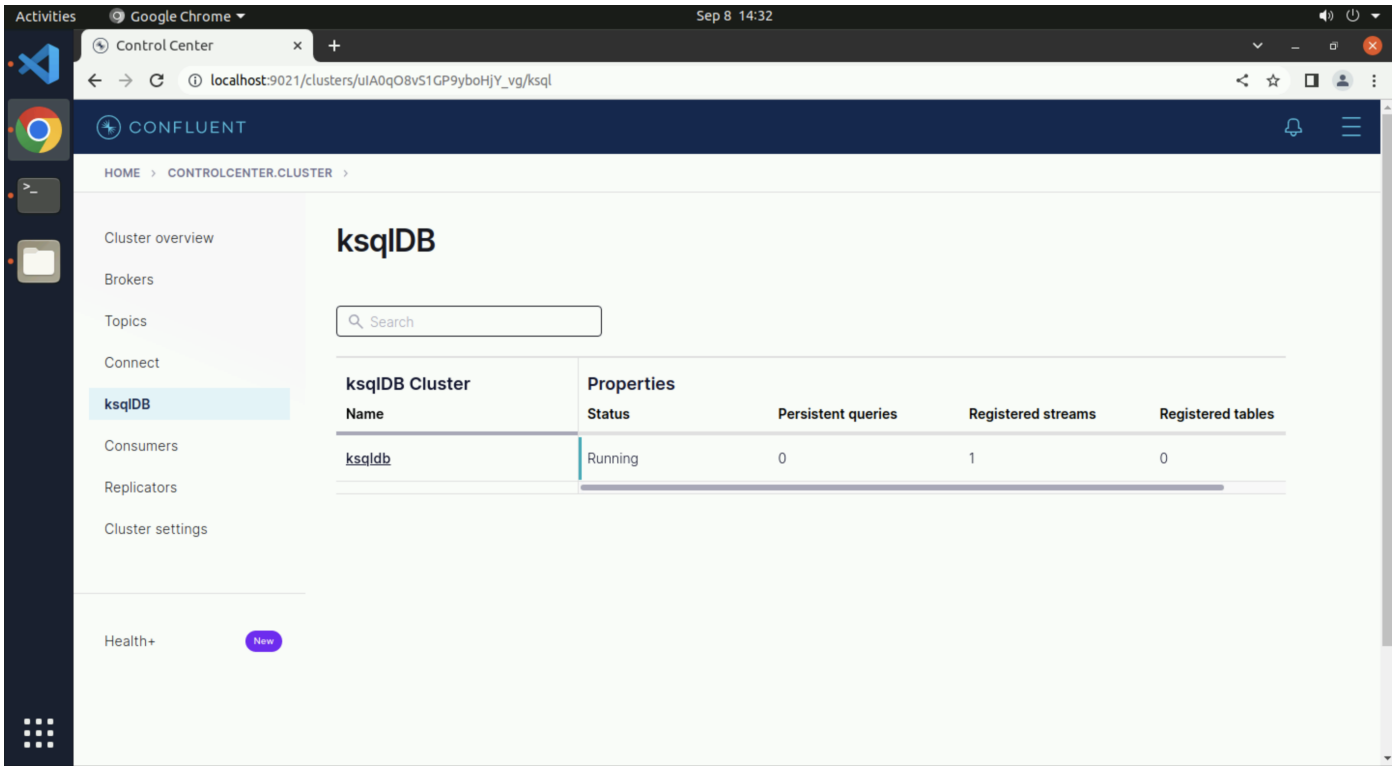


Here we get a peek into the inflowing records. Please familiarize yourself with this view. Notice that you have 2 display modes for the records, tabular and card view. Experiment with them. Also notice the **Topic Summary** on the left side of the view, listing message fields and other information about the topic.

6. Now click on the tab **Schema**. You will not see anything meaningful here since we are not using **AVRO, Protobuf, or JSON** as data format in our sample. If we used one of these, then this view would show you schema details and version(s).

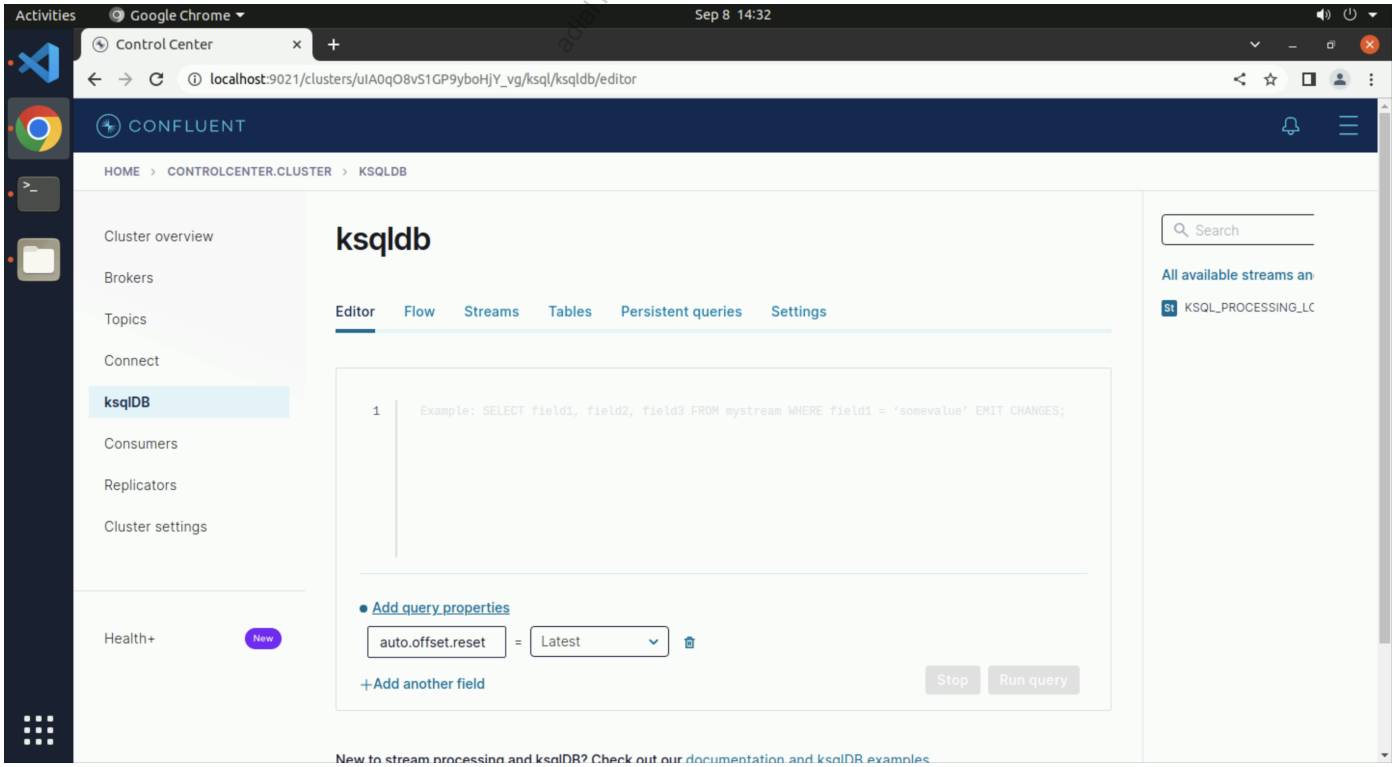
Using ksqlDB for simple real-time stream transformations

1. From the list of tabs on the left select **ksqlDB**. You should see this:



This is the list of connected ksqlDB clusters. Currently we only have one cluster called **ksqldb**.

2. Select this **ksqldb** cluster. You should see this:

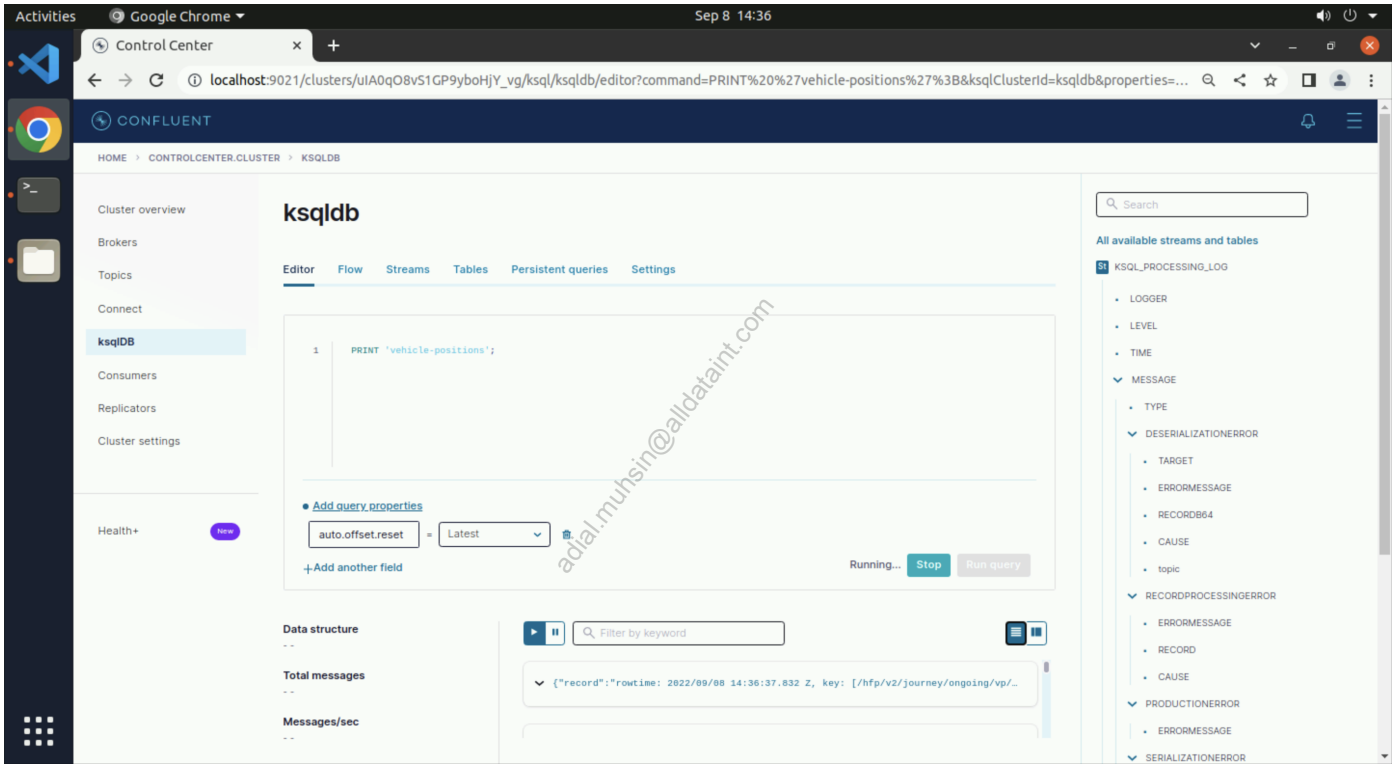


Notice that this is a tabbed view. Currently we are on the **ksqlDB Editor** tab.

3. In the ksqlDB editor field of this view enter `SHOW topics;` and then click the button **Run**. In the result area you should see a list of all topics that are defined in Kafka. The last topic in the list should be our `vehicle-positions` topic.

4. Now enter `PRINT 'vehicle-positions';` in the ksqlDB editor field and then click the **Run** button. You should get a raw output of the content of the topic `vehicle-positions`

The single quotes are vital to prevent the hyphen from causing an error.



Click the button **Stop** when done.

5. Now let's define a **stream** from the topic `vehicle-positions`. In the ksqlDB editor field enter the following query:

```
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');
```


and then click the **Run** button. ksqlDB will create a stream called **`VEHICLE_POSITIONS`**. You should see an output generated similar to:

```
{
  "@type": "currentStatus",
  "statementText": "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',
KEY_FORMAT='KAFKA', VALUE_FORMAT='JSON');",
  "commandId": "stream/`VEHICLE_POSITIONS`/create",
  "commandStatus": {
    "status": "SUCCESS",
    "message": "Stream created",
    "queryId": null
  },
  "commandSequenceNumber": 2,
  "warnings": [

  ]
}
```


6. Now we want to use the query data from the stream we just created. In the ksqlDB editor field enter the following query:


```
SELECT vp->desi, vp->dir FROM VEHICLE_POSITIONS EMIT CHANGES;
```

and we will see an output similar to this (This may take a few seconds to populate the table on the lower right corner of the screen):

```
717A | 1
N | 1
736 | 1
18 | 2
203N | 2
172 | 1
975 | 2
841 | 2
55 | 1
443 | 1
```

7. Press **Stop** to stop to end the query.

8. Now let's try to filter the data and only show records for the bus number [600: Rautatientori - Lentoasema]. In the ksqlDB editor field enter the following query:

```
SELECT * FROM VEHICLE_POSITIONS
      WHERE vp->desi='600'
      EMIT CHANGES;
```

Now we get only output with a route number of `'600'`.

9. Press **Stop** to stop to end the query.

Cleanup

1. Before you end, please clean up your system by running the `stop.sh` script in the project folder from your labVM Terminal window:

```
$ cd ~/confluent-fundamentals/labs/confluent-platform
$ ./stop.sh
```

Conclusion

In this lab we used Confluent Control Center to monitor our Kafka cluster. We used it to inspect the topic `vehicle-positions`, specifically the messages that flow into the topic and their schema. We also observed the consumer lag. Finally, we used ksqlDB to do simple real-time stream transformations.

STOP HERE. THIS IS THE END OF THE EXERCISE.