Kafka Connect Workshop

Table of Contents

Lab 1: Connecting to your Workshop Environment

Your environment represents an on-premise data center and consists of a virtual machine hosted in the cloud running several docker containers. In a real world implementation, some of the components would be deployed differently but the logical data flow that we will be working on would remain the same. You will use a setup that includes a virtual machine in AWS (EC2) configured as a Docker host that will include all the required software to perform the exercises.

To log in to your virtual data center open a terminal session and use the credentials that were assigned to you.

NOTE: if you are using docker-compose in your computer you don't need to connect with the following ssh command, you will only need to run the commands that are part of the labs locally.

```
ssh dc16@13.40.228.238
```

Once logged in run the following command to confirm that you have several docker containers running

```
docker ps --format "table {{.ID}}\t{{.Names}}\t{{.RunningFor}}\t{{.Status}}"
```

You should see something similar to this.

```
CONTAINER ID   NAMES                    CREATED              STATUS
634ebca7c9cc   control-center           40 seconds ago       Up 38 seconds
0753e6c7c452   ksqldb-cli               40 seconds ago       Up 37 seconds
b9e35166dc92   ksqldb-server            42 seconds ago       Up 40 seconds
797f867c6d0a   connect                  43 seconds ago       Up 42 seconds (health: starting)
9016da9bc202   rest-proxy               43 seconds ago       Up 38 seconds
e3af8d77c3f8   schema-registry          45 seconds ago       Up 43 seconds
4c028b5367a7   mongo-express            47 seconds ago       Up 46 seconds
79a1c2a11dab   broker                   49 seconds ago       Up 45 seconds
f707b1ed252f   zookeeper                About a minute ago   Up 48 seconds
9ef66a7822c7   postgres-db              About a minute ago   Up 52 seconds
adb07eef2aa7   mymongodb                About a minute ago   Up 47 seconds
42ecfd6c153b   pg_dashboard             About a minute ago   Up 49 seconds
913a296c4ba9   workshop-docs-webserver  About a minute ago   Up 51 seconds
```

Whenever you see this icon it means the step is mandatory. Missing one of these steps will result in the data pipeline not working as expected.

Take advantage of the copy to clipboard button, it will save you some time!

Lab 2: Getting Started

The primary data source for this workshop is a Postgres database running in your data center which has a schema with one table called *Customers*. We will be able during the following labs to take the *Customers* data from the Postgres database and send it to a Confluent Platform Cluster using Kafka Connect and leveraging the amazing portfolio of more than 120 pre-built connectors available. Applying simple transformations to messages as they flow through Connect using Single Message Transforms in the connectors configurations. Once the customer data is available in the Confluent Platform Cluster you could enrich or/and process it with SQL lightweight syntax using ksqlDB making it available for

other systems. In this case we are going to do a database modernization sending the data to a MongoDB database using the available Sink connectors.

Connect to the database

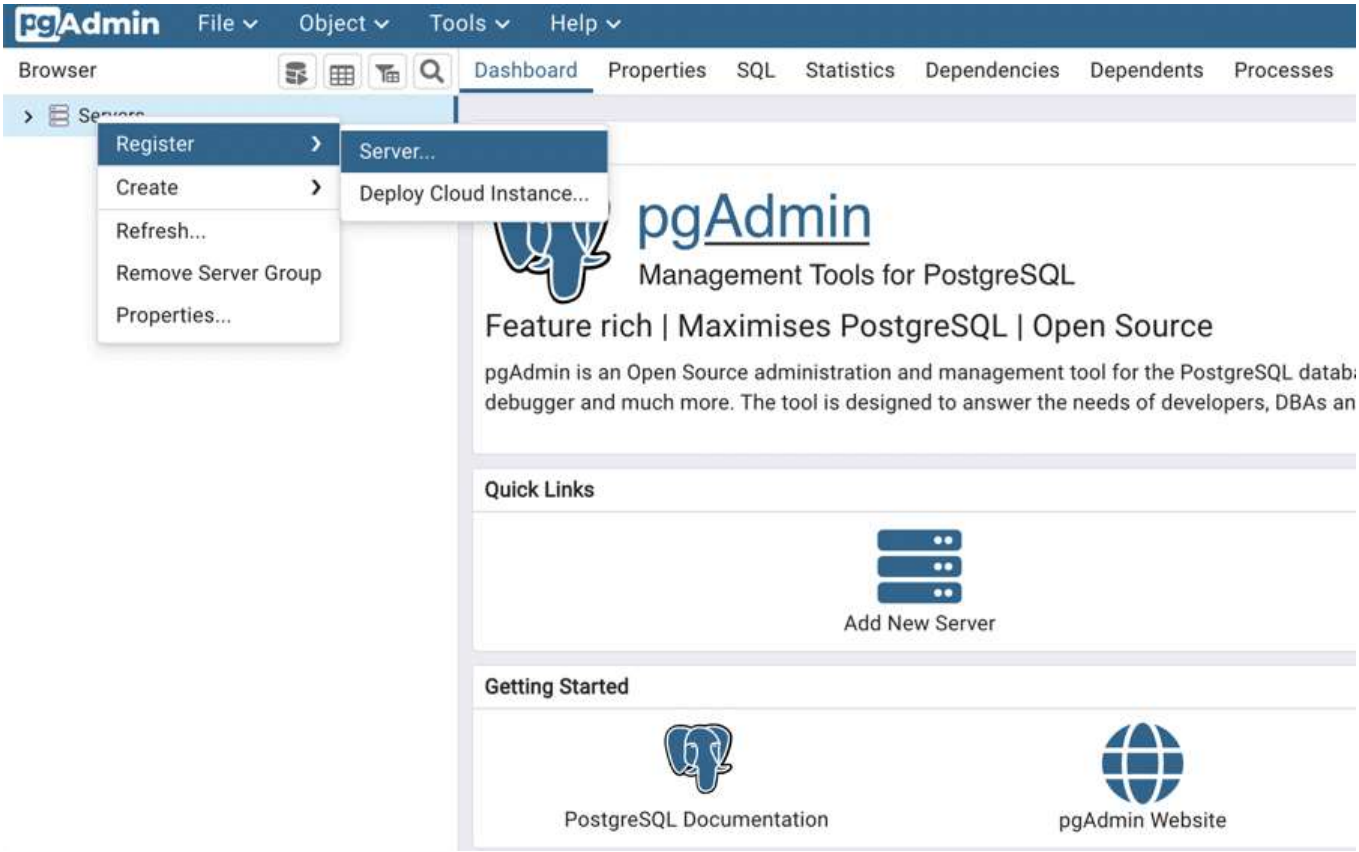
The environment has PgAdmin UI installed so we will be able check *Customers* data from the Postgres database using the browser.

Please access using to PgAdmin UI

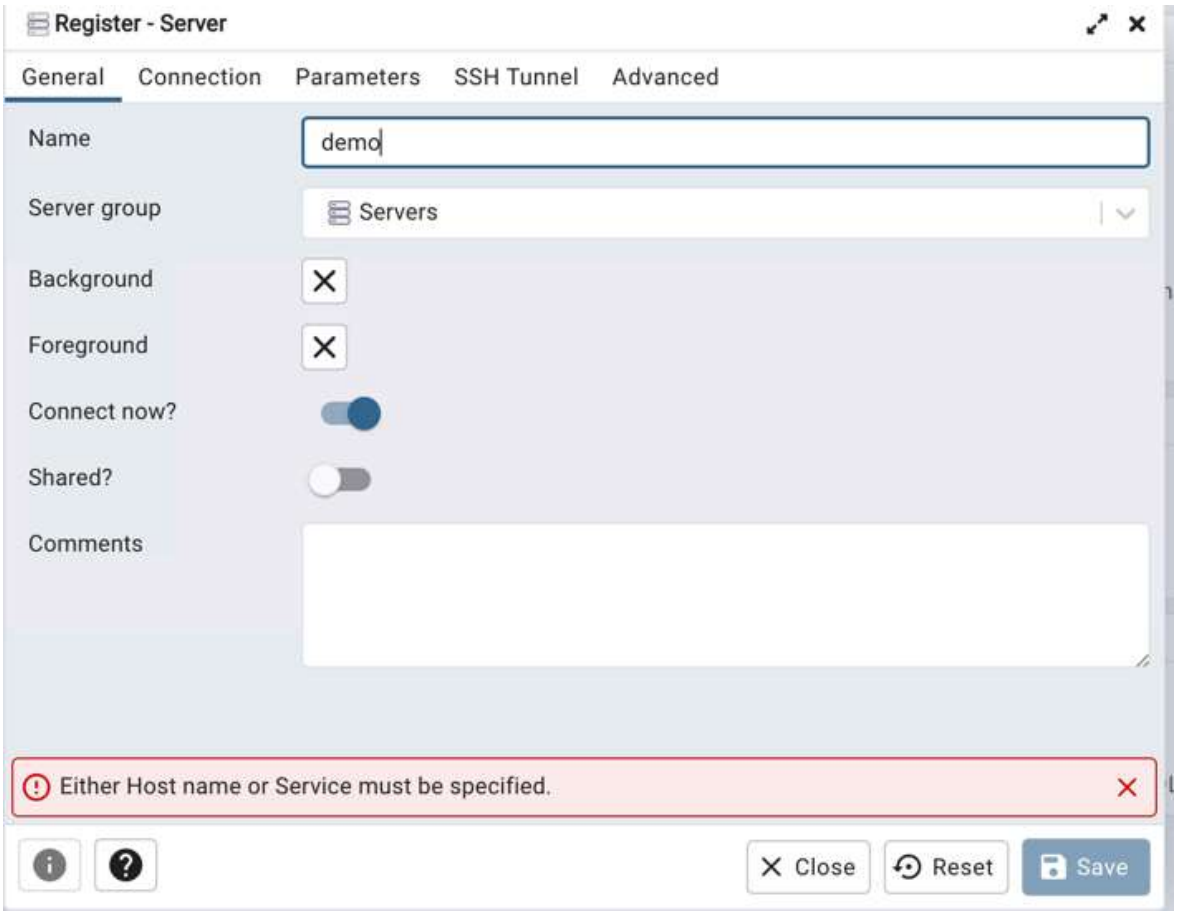
Use the following and username and password to authenticate:

```
Username: dc16@mail.com
Password: your workshop password
```

After logging into PgAdmin UI configure the connection to the database. Go to the left menu called *Server*, right click and select *Register* and select *Server...* as shown in the following picture:

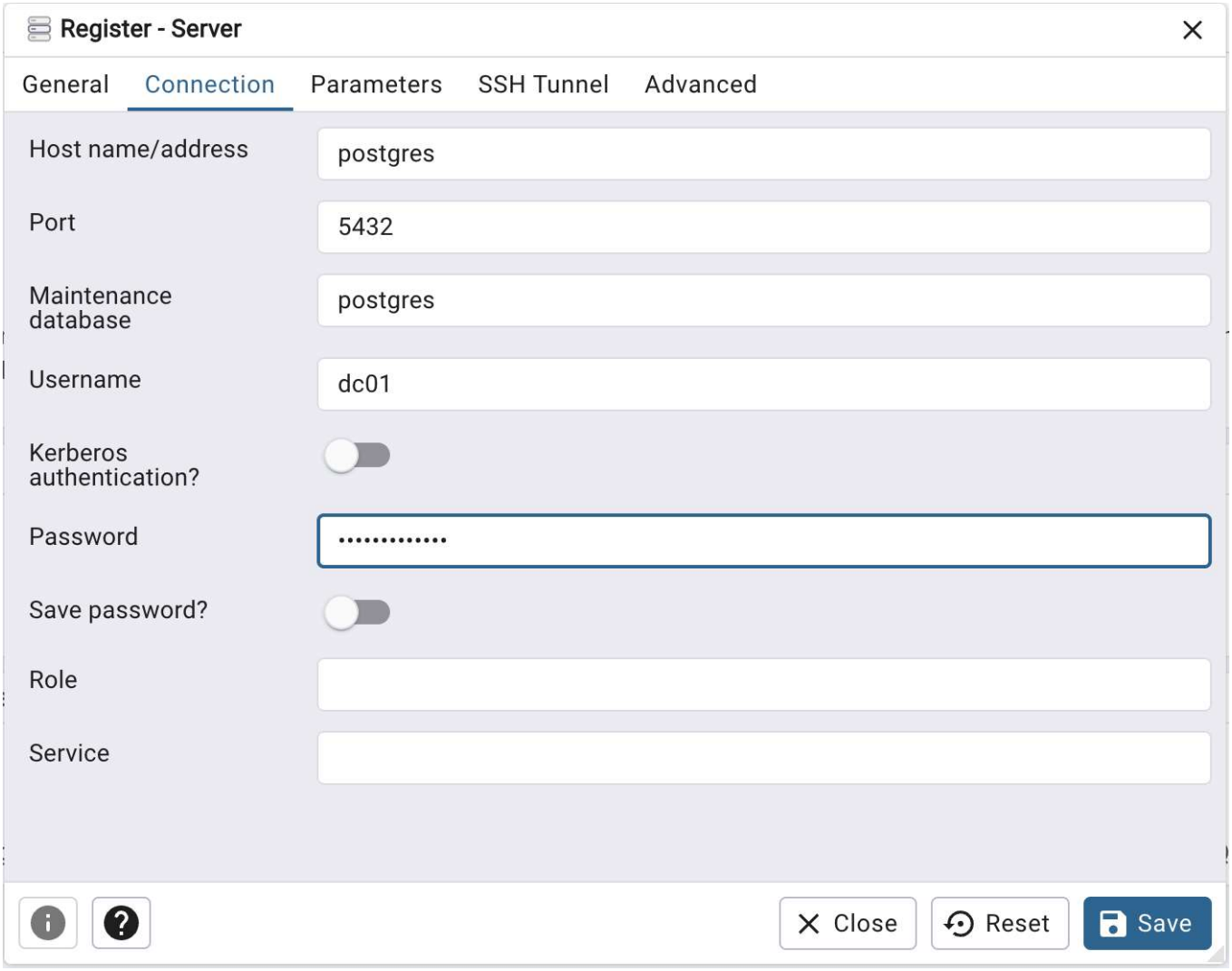


Configure the database following the steps in the pictures:

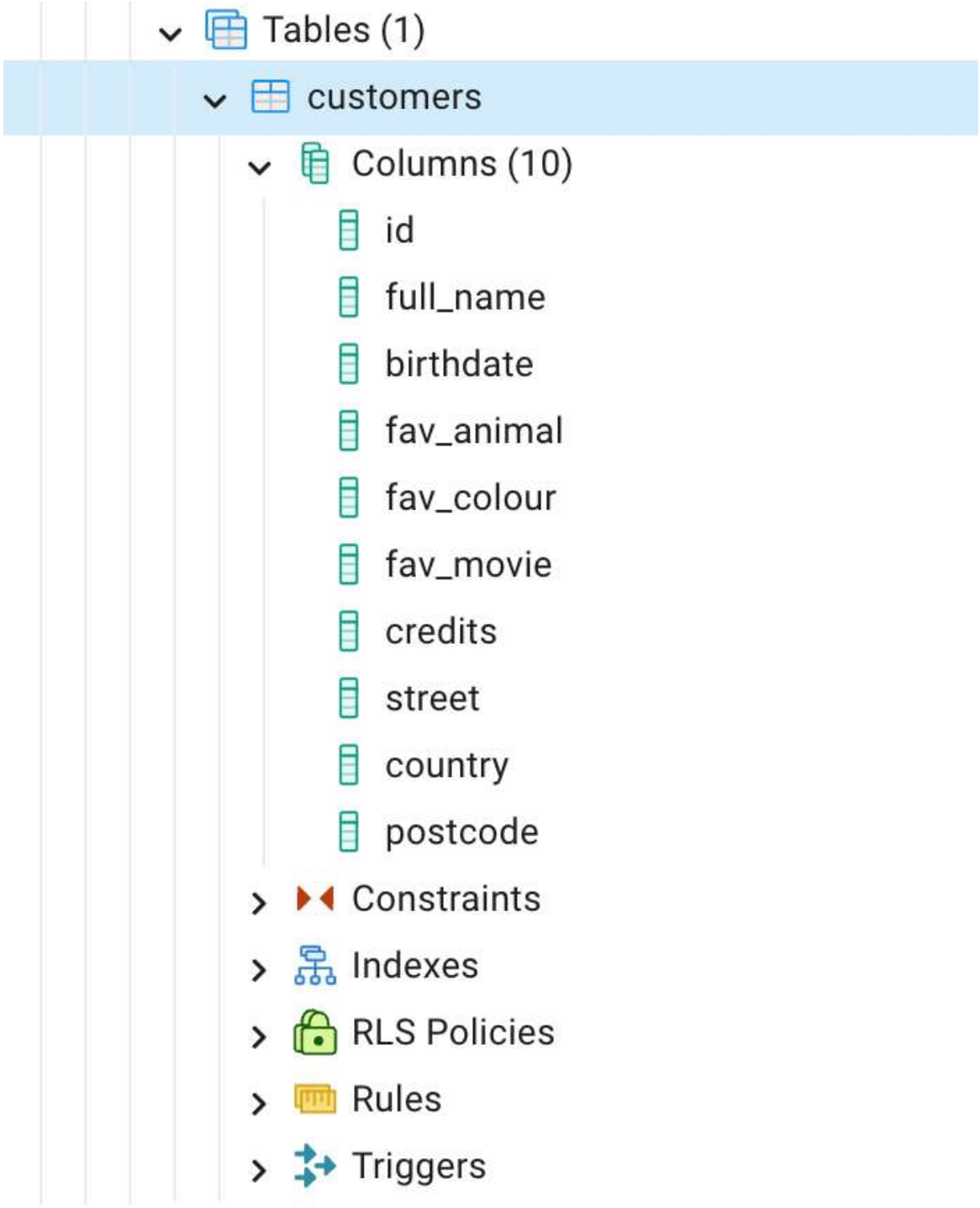


Use the following and username and password to authenticate to the database:

```
Hostname: postgres
Maintenance database: postgres
User: dc16
Password: your workshop password
```



Once the connection to the database has been successful, we will be able to find the the *Public* Schema inside the *postgres* database which will have a *Customers* table:

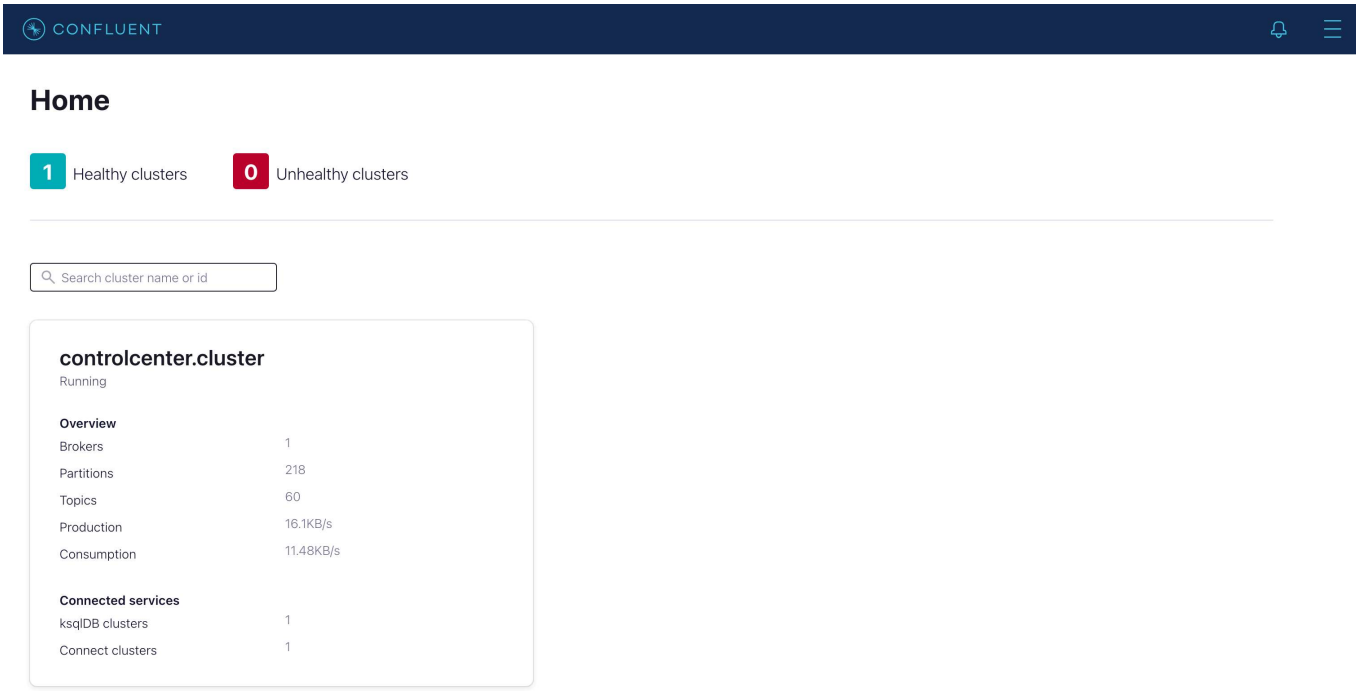


Check your access to Confluent Control Center

Now that the Postgres is configured, let's check if we have access to our local Kafka cluster.

We can use Confluent Control Center to confirm this. Use your user dc16 and your workshop password to log in.

On the landing page we can see that Confluent Control Center is monitoring our Kafka Cluster.



Lab 3: Source data from Postgres to Confluent Platform

To migrate the data from Postgres database to Confluent we are going to leverage the complete connect portfolio that is already pre-built.

In this case we are going to use PostgreSQL Source Connector (Debezium). To read from *Customers* table we will execute the following cURL command using the shh connection to the environment that we opened before, we are just migrating two columns from the table (id and full_name):

```
curl -i -X PUT -H "Accept:application/json" \
-H  "Content-Type:application/json" http://localhost:8083/connectors/postgres-source-connector-customer00/config \
-d '{
               "connector.class": "io.debezium.connector.postgresql.PostgresConnector",
               "tasks.max": "1",
               "database.hostname": "db",
               "database.port": "5432",
               "database.user": "dc16",
               "database.password": "'${PASS}'",
               "database.dbname" : "postgres",
               "schema.include.list": "public",
               "table.include.list": "public.customers",
               "column.include.list": "public.customers.id,public.customers.full_name",
               "topic.prefix": "postgres00_",
               "plugin.name": "pgoutput",
```

```
                "provide.transaction.metadata": false,
                "slot.name" : "0"

        }'
```

You should see an output like this:

```
HTTP/1.1 201 Created
Date: Thu, 30 Mar 2023 11:41:10 GMT
Location: http://localhost:8083/connectors/postgres-source-connector-customer00
Content-Type: application/json
Content-Length: 439
Server: Jetty(9.4.44.v20210927)

{"name":"postgres-source-connector-customer00","config":
{"connector.class":"io.debezium.connector.postgresql.PostgresConnector","tasks.max":"1","database.
hostname":"db","database.port":"5432","database.user":"postgres","database.password":"postgres","d
atabase.dbname":"postgres","schema.include.list":"public","table.include.list":"city","topic.prefi
x":"postgres_","plugin.name":"pgoutput","name":"postgres-source-connector"},"tasks":
[],"type":"source"}
```

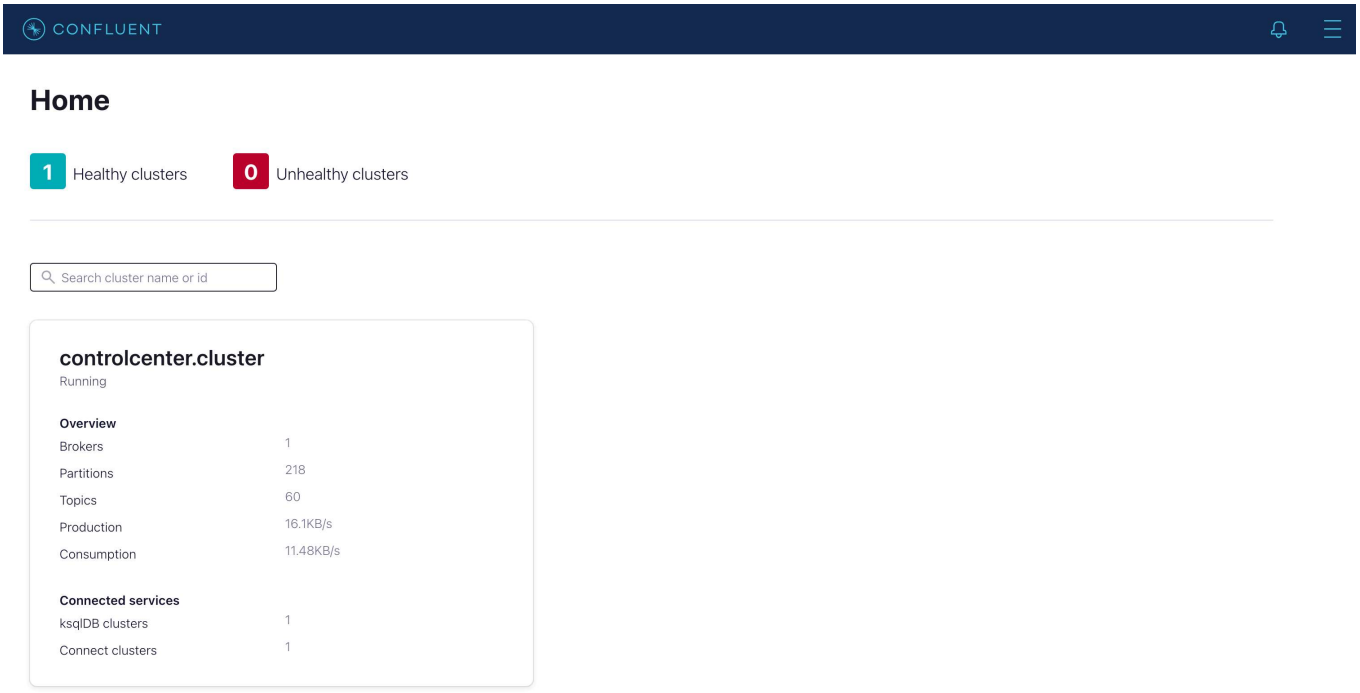
We can confirm the connector is running by querying the REST interface

```
curl -s localhost:8083/connectors/postgres-source-connector-customer00/status | jq
```

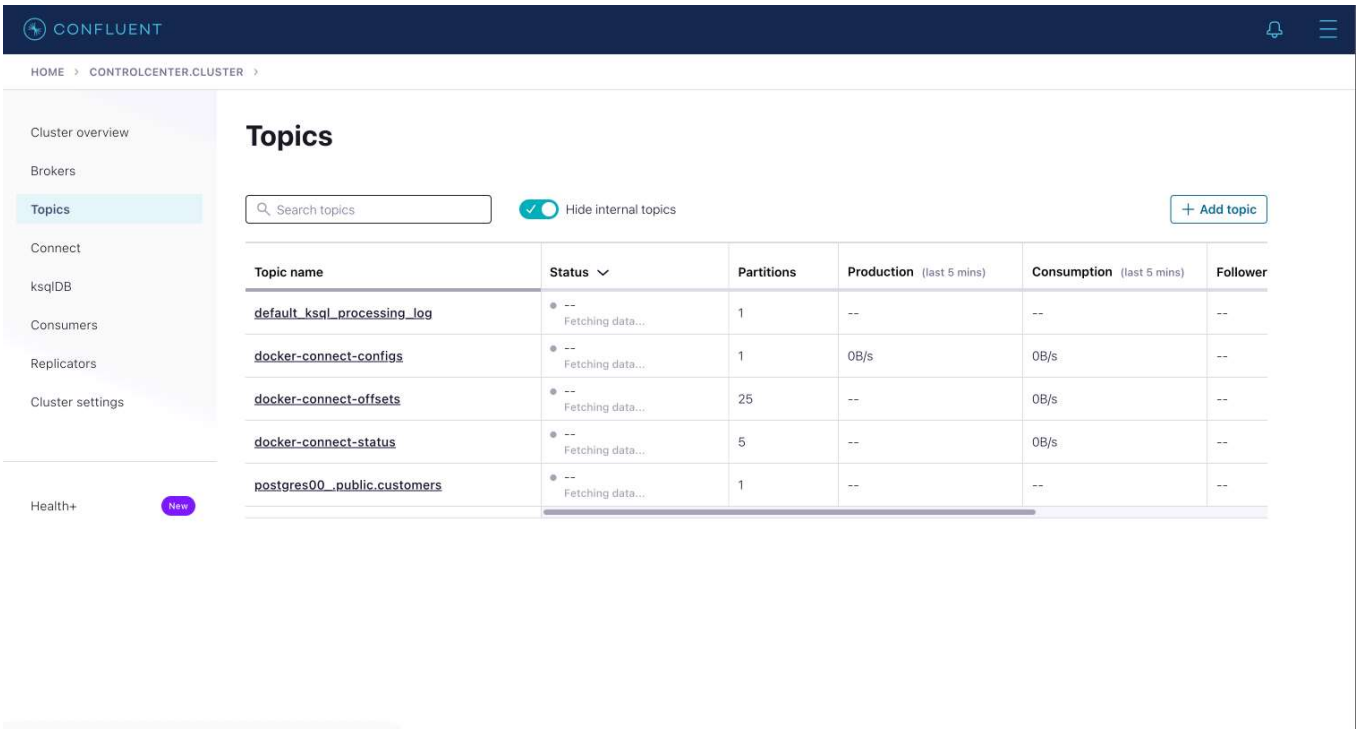
Now that the PostgreSQL Source Connector is up and running, we will be able to see messages appearing in our local Kafka cluster.

As we are already logged in Confluent Control Center we will be able to confirm this.

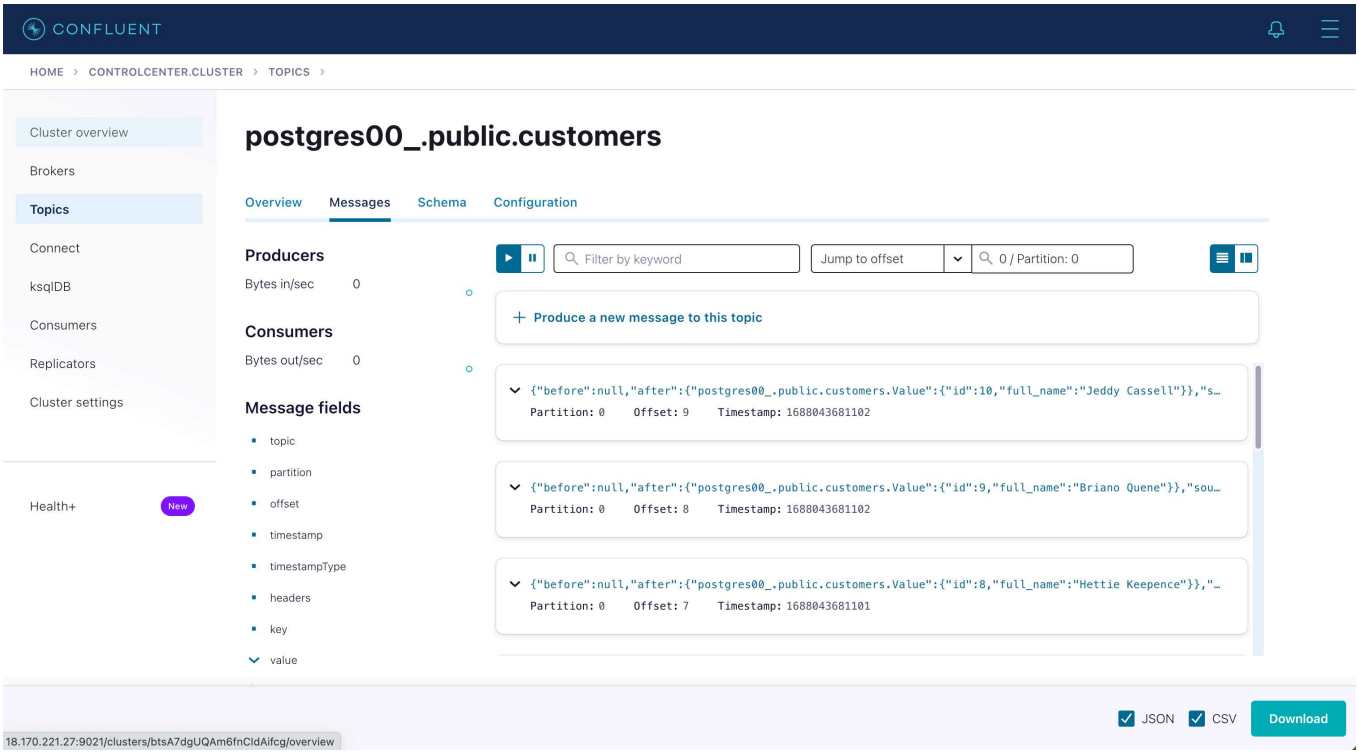
Click on the "controlcenter.cluster" tile, this is your on-premise cluster:



Select the **Topics Menu** on the left:



Select the `postgres00_.public.customers` topic and select the **Messages tab** and observe that messages are being streamed into Kafka from Postgres in real time. **Please put 0 in the Jump to offset option in the UI to be able to see the messages from the very first offset**



If we inspect one message value:


```
{
  "before": null,
  "after": {
    "postgres00_.public.customers.Value": {
      "id": 10,
      "full_name": "Jeddy Cassell"
    }
  },
  "source": {
    "version": "2.2.1.Final",
    "connector": "postgresql",
    "name": "postgres00_",
    "ts_ms": 1688395265137,
    "snapshot": {
      "string": "last"
    },
    "db": "postgres",
    "sequence": {
      "string": "[null,\"24251064\"]"
    },
    "schema": "public",
    "table": "customers",
    "txId": {
      "long": 748
    },
    "lsn": {
      "long": 24251064
```

```
      },
      "xmin": null
    },
    "op": "r",
    "ts_ms": {
      "long": 1688395265337
    },
    "transaction": null
  }
```

Notice that it has the data **postgres00_.public.customers.Value** and metadata added after the value.

Also if we can check if the message has key, in this case is null:

Further Reading

- Debezium Postgres Source Connector

Lab 4: Single Message Transforms - ValueToKey and ExtractNewRecordState

We could see in the previous picture that we had messages without key in the topic that the connector created in the previous step. We want to have a key in our messages in order to have them correctly ordered within the topic partitions. We can achieve that adding SMT configs to the previous connector, **ValueToKey** which will help us to have a proper key in the messages.

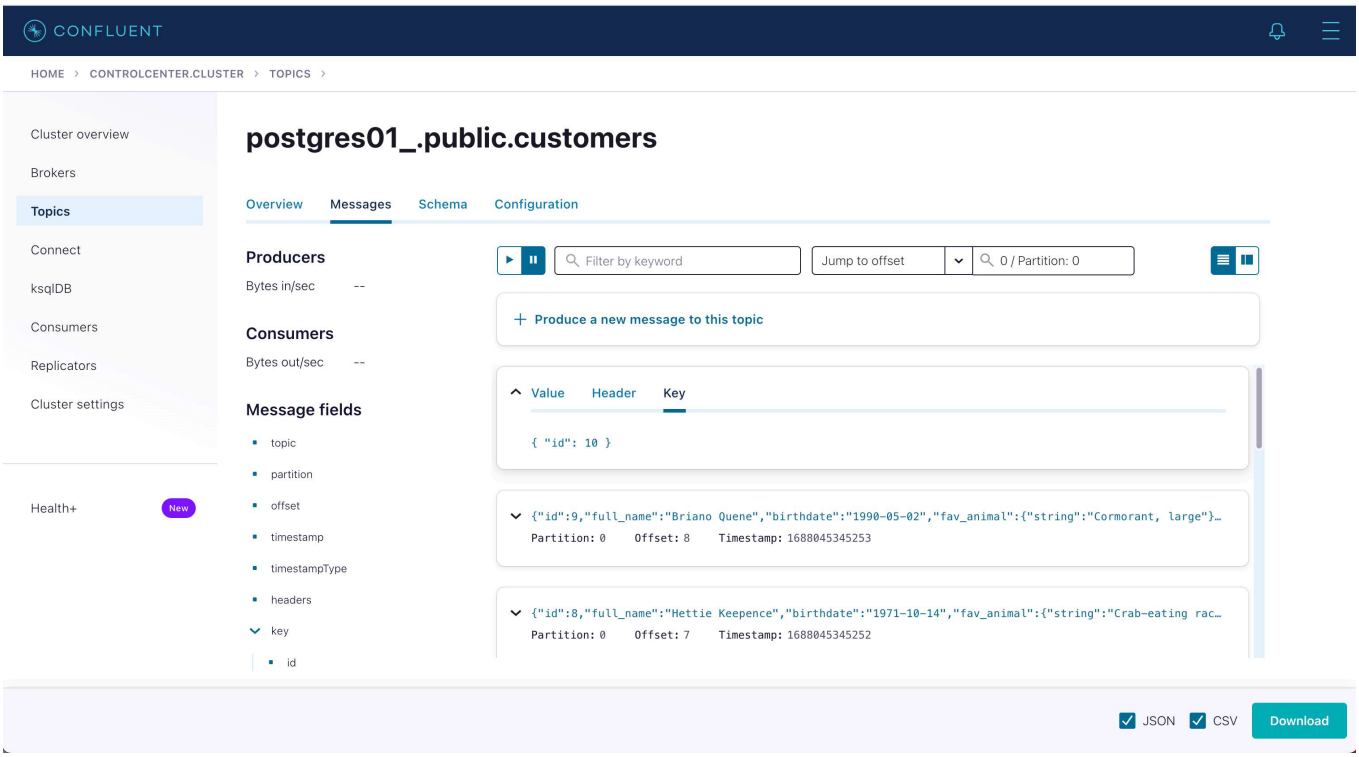
And as we saw in the message that we inspected earlier, it had lots of metadata, we also can keep just the metadata fields that we want using **ExtractNewRecordState.**

Execute the following cURL command:

```
curl -i -X PUT -H "Accept:application/json" \
-H  "Content-Type:application/json" http://localhost:8083/connectors/postgres-source-connector-customer01/config \
-d '{
                "connector.class":
"io.debezium.connector.postgresql.PostgresConnector",
                "tasks.max": "1",
                "database.hostname": "db",
```

```
                "database.port": "5432",
                "database.user": "dc16",
                "database.password": "'${PASS}'",
                "database.dbname" : "postgres",
                "schema.include.list": "public",
                "table.include.list": "public.customers",
                "topic.prefix": "postgres01_",
                "plugin.name": "pgoutput",
                "transforms": "extract,createkey",
                "transforms.extract.type":
"io.debezium.transforms.ExtractNewRecordState",
                "transforms.extract.add.fields": "op,table,source.ts_ms",
                "transforms.extract.drop.tombstones": "false",
                "transforms.extract.delete.handling.mode": "rewrite",
                "transforms.createkey.type":
"org.apache.kafka.connect.transforms.ValueToKey",
                "transforms.createkey.fields": "id",
                "slot.name" : "1"
    }'
```

Check in Confluent Control Center if the messages in the topic `postgres01.public.customers` have a key selecting, once you have already selected one of the messages, the **key** tab:



As per this lab purposes we are creating different connectors (the names are different in the cURL commands that we execute) but if you want to replace the first one and ensure that it is working with the new configurations, you will need to have new data as it will be only applied to new messages. So you can add new data to customers table using the PgAdmin UI and check the connector results:

```
INSERT INTO customers (id, full_name, birthdate, fav_animal, fav_colour, fav_movie, street,
country, postcode)
VALUES (11, 'Sam Smith', '1990-02-06', 'Mouse', 'Puce', 'The notebook',
'Lynchburg','Virginia','24515');

SELECT id, full_name, birthdate, fav_animal, fav_colour, fav_movie
    FROM public.customers;
```

Further Reading

- Valuetokey
- ExtractNewRecordState

Lab 5: Single Message Transforms - ReplaceField

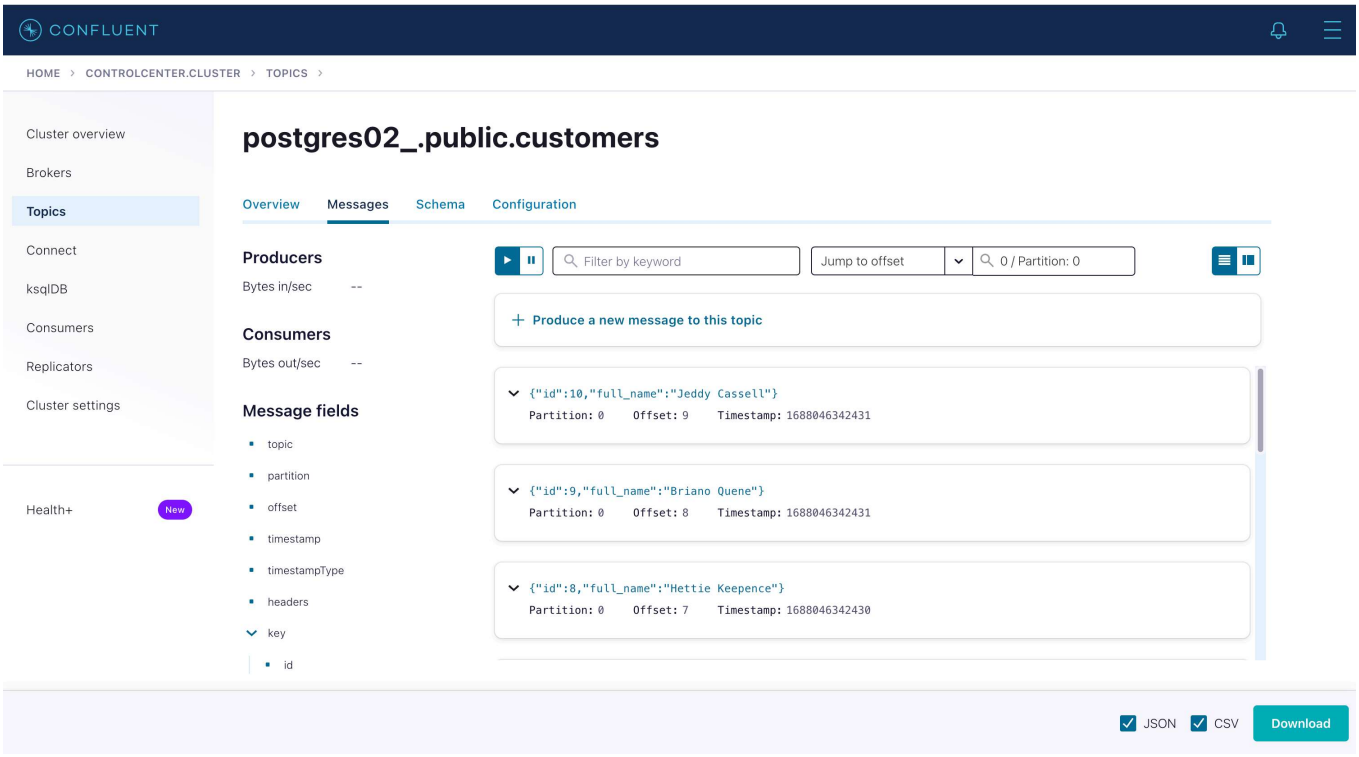
The very first connector that we created was selecting just some columns from the table customers using this configuration: `"column.include.list": "public.customers.id,public.customers.full_name"`.

It is possible to achieve the same result using the SMT **ReplaceField$Value.**

```
curl -i -X PUT -H "Accept:application/json" \
-H  "Content-Type:application/json" http://localhost:8083/connectors/postgres-
source-connector-customer02/config \
-d '{
               "connector.class":
"io.debezium.connector.postgresql.PostgresConnector",
               "tasks.max": "1",
               "database.hostname": "db",
               "database.port": "5432",
               "database.user": "dc16",
               "database.password": "'${PASS}'",
               "database.dbname" : "postgres",
               "schema.include.list": "public",
               "table.include.list": "public.customers",
               "topic.prefix": "postgres02_",
               "plugin.name": "pgoutput",
               "transforms": "extract,createkey,selectFields",
               "transforms.extract.type":
"io.debezium.transforms.ExtractNewRecordState",
               "transforms.createkey.type":
"org.apache.kafka.connect.transforms.ValueToKey",
               "transforms.createkey.fields": "id",
               "transforms.selectFields.type"     :
"org.apache.kafka.connect.transforms.ReplaceField$Value",
```

```
                "transforms.selectFields.include"  : "id,full_name",
                "slot.name" : "2"
}'
```

Check in Confluent Control Center if the messages in the topic `postgres02.public.customers` just have two fields (id,full_name):



- Replacefield

Lab 6: Single Message Transforms - ExtractField and ValueToKey

The SMTs used in the previous connectors write a struct to the key, and often we want just the primitive value instead.

That's what combining **ExtractField$Key** and **ValueToKey** do.

```
curl -i -X PUT -H "Accept:application/json" \
-H  "Content-Type:application/json" http://localhost:8083/connectors/postgres-source-connector-customer03/config \
-d '{
                "connector.class": "io.debezium.connector.postgresql.PostgresConnector",
                "tasks.max": "1",
                "database.hostname": "db",
                "database.port": "5432",
```

```
                "database.user": "dc16",
                "database.password": "'${PASS}'",
                "database.dbname" : "postgres",
                "schema.include.list": "public",
                "table.include.list": "public.customers",
                "topic.prefix": "postgres03_",
                "plugin.name": "pgoutput",
                "transforms": "extract,createkey,extractKeyFromStruct",
                "transforms.extract.type": "io.debezium.transforms.ExtractNewRecordState",
                "transforms.createkey.type": "org.apache.kafka.connect.transforms.ValueToKey",
                "transforms.createkey.fields": "id",

"transforms.extractKeyFromStruct.type":"org.apache.kafka.connect.transforms.ExtractField$Key",
                "transforms.extractKeyFromStruct.field": "id",
                "slot.name" : "3"
}'
```

Check using ksqlDB console if the messages in the topic `postgres03_.public.customers` has a primitive value in their key:

```
docker exec -it ksqldb-cli ksql http://ksqldb-server:8088
```

You should see something like this:-

```
                ===========================================
                =       _              _ ____  ____       =
                =      | | _____  __ _| |  _ \| __ )      =
                =      | |/ / __|/ _` | | | | |  _ \      =
                =      |   <\__ \ (_| | | |_| | |_) |     =
                =      |_|\_\___/\__, |_|____/|____/      =
                =                   |_|                   =
                =  Event Streaming Database purpose-built =
                =        for stream processing apps       =
                ===========================================

Copyright 2017-2022 Confluent Inc.

CLI v7.3.0, Server v7.3.0 located at http://ksqldb-server-ccloud:8088

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

ksql>
```

Once you are connected execute the following command:

```
print `postgres03_.public.customers` from beginning;
```

You should see the following output, please take a look to the **key** field:

```
Key format: AVRO or KAFKA_STRING
Value format: AVRO or KAFKA_STRING
rowtime: 2023/06/29 13:54:00.569 Z, key: 1, value: {"id": 1, "full_name": "Leone Puxley",
"birthdate": "1995-02-06", "fav_animal": "Violet-eared waxbill", "fav_colour": "Puce",
"fav_movie": "Oh! What a Lovely War", "credits": "53.49", "street": "Lynchburg", "country":
"Virginia", "postcode": "24515"}, partition: 0
rowtime: 2023/06/29 13:54:00.570 Z, key: 2, value: {"id": 2, "full_name": "Angelo Sharkey",
"birthdate": "1996-04-08", "fav_animal": "Macaw, green-winged", "fav_colour": "Red", "fav_movie":
"View from the Top, A", "credits": "7.0", "street": "Manassas", "country": "Virginia", "postcode":
"22111"}, partition: 0
rowtime: 2023/06/29 13:54:00.570 Z, key: 3, value: {"id": 3, "full_name": "Jozef Bailey",
"birthdate": "1954-07-10", "fav_animal": "Little brown bat", "fav_colour": "Indigo", "fav_movie":
"99 francs", "credits": "5.49", "street": "Lexington", "country": "Kentucky", "postcode":
"40515"}, partition: 0
rowtime: 2023/06/29 13:54:00.570 Z, key: 4, value: {"id": 4, "full_name": "Evelyn Deakes",
"birthdate": "1975-09-13", "fav_animal": "Vervet monkey", "fav_colour": "Teal", "fav_movie": "Jane
Austen in Manhattan", "credits": "8.09", "street": "Chicago", "country": "Illinois", "postcode":
"60681"}, partition: 0
rowtime: 2023/06/29 13:54:00.571 Z, key: 5, value: {"id": 5, "full_name": "Dermot Perris",
"birthdate": "1991-01-29", "fav_animal": "African ground squirrel (unidentified)", "fav_colour":
"Khaki", "fav_movie": "Restless", "credits": "3.49", "street": "Asheville", "country": "North
Carolina", "postcode": "28805"}, partition: 0
rowtime: 2023/06/29 13:54:00.571 Z, key: 6, value: {"id": 6, "full_name": "Renae Bonsale",
"birthdate": "1965-01-05", "fav_animal": "Brown antechinus", "fav_colour": "Fuscia", "fav_movie":
"Perfect Day, A (Un giorno perfetto)", "credits": "77.40", "street": "San Jose", "country":
"California", "postcode": "95113"}, partition: 0
rowtime: 2023/06/29 13:54:00.571 Z, key: 7, value: {"id": 7, "full_name": "Florella Fridlington",
"birthdate": "1950-08-07", "fav_animal": "Burmese brown mountain tortoise", "fav_colour":
"Purple", "fav_movie": "Dot the I", "credits": "50.0", "street": "Jamaica", "country": "New York",
"postcode": "11431"}, partition: 0
rowtime: 2023/06/29 13:54:00.571 Z, key: 8, value: {"id": 8, "full_name": "Hettie Keepence",
"birthdate": "1971-10-14", "fav_animal": "Crab-eating raccoon", "fav_colour": "Puce", "fav_movie":
"Outer Space", "credits": "4.0", "street": "Pensacola", "country": "Florida", "postcode":
"32590"}, partition: 0
rowtime: 2023/06/29 13:54:00.572 Z, key: 9, value: {"id": 9, "full_name": "Briano Quene",
"birthdate": "1990-05-02", "fav_animal": "Cormorant, large", "fav_colour": "Yellow", "fav_movie":
"Peacekeeper, The", "credits": "3.0", "street": "San Antonio", "country": "Texas", "postcode":
"78296"}, partition: 0
rowtime: 2023/06/29 13:54:00.572 Z, key: 10, value: {"id": 10, "full_name": "Jeddy Cassell",
"birthdate": "1978-12-24", "fav_animal": "Badger, european", "fav_colour": "Indigo", "fav_movie":
"Shadow of a Doubt", "credits": "2.0", "street": "Charleston", "country": "West Virginia",
"postcode": "25331"}, partition: 0
```

Remember to **exit** from the ksqlBD console to continue with the following lab.

- Extractfield
- Valuetokey

Lab 7: Single Message Transforms - Filter$Value

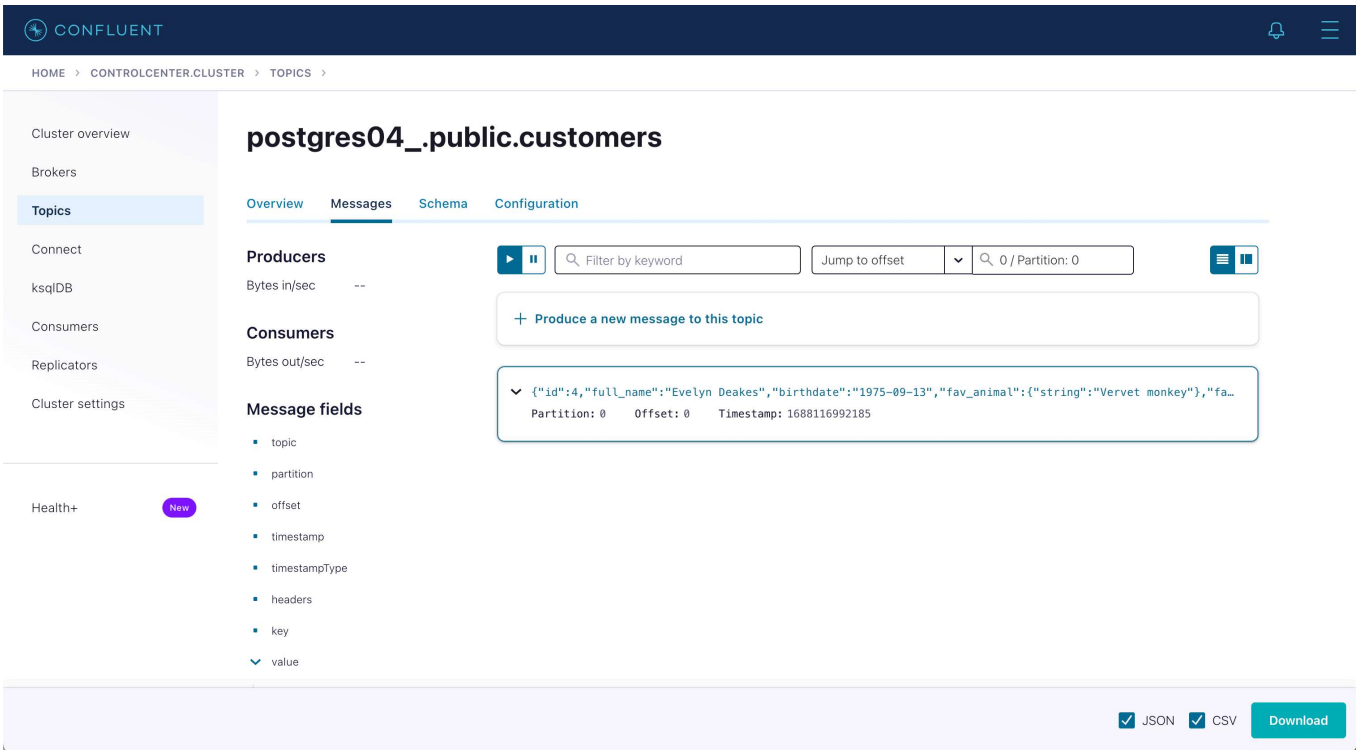
SMT lets us also filter messages before inserting them into kafka and that is possible using Confluent **Filter$Value** which filters based on the message content. And you have the option to include or exclude the messages that meet the condition.

By executing the following command we are including those that meet the condition:

```
curl -i -X PUT -H "Accept:application/json" \
-H  "Content-Type:application/json" http://localhost:8083/connectors/postgres-source-
connector-customer04/config \
-d '{
                "connector.class": "io.debezium.connector.postgresql.PostgresConnector",
                "tasks.max": "1",
                "database.hostname": "db",
                "database.port": "5432",
                "database.user": "dc16",
                "database.password": "'${PASS}'",
                "database.dbname" : "postgres",
                "schema.include.list": "public",
                "table.include.list": "public.customers",
                "topic.prefix": "postgres04_",
                "plugin.name": "pgoutput",
                "transforms": "extract,createkey,extractKeyFromStruct,filterExample",
                "transforms.extract.type": "io.debezium.transforms.ExtractNewRecordState",
                "transforms.createkey.type": "org.apache.kafka.connect.transforms.ValueToKey",
                "transforms.createkey.fields": "id"  ,

"transforms.extractKeyFromStruct.type":"org.apache.kafka.connect.transforms.ExtractField$Key",
                "transforms.extractKeyFromStruct.field":"id",
                "transforms.filterExample.type":
"io.confluent.connect.transforms.Filter$Value",
                "transforms.filterExample.filter.condition": "$[?(@.fav_animal =~
/.*monkey/)]",
                "transforms.filterExample.filter.type": "include",
                "slot.name" : "4"
}'
```

Check in Confluent Control Center if the messages in the topic `postgres04.public.customers` are just the ones that has as part of fav_animal value: monkey.



- Filter

Lab 8: Single Message Transforms - Filter$Value and Cast$Value

We can filter on numerics too, we need to make sure that the data type is correct using SMT **Cast$Value.**

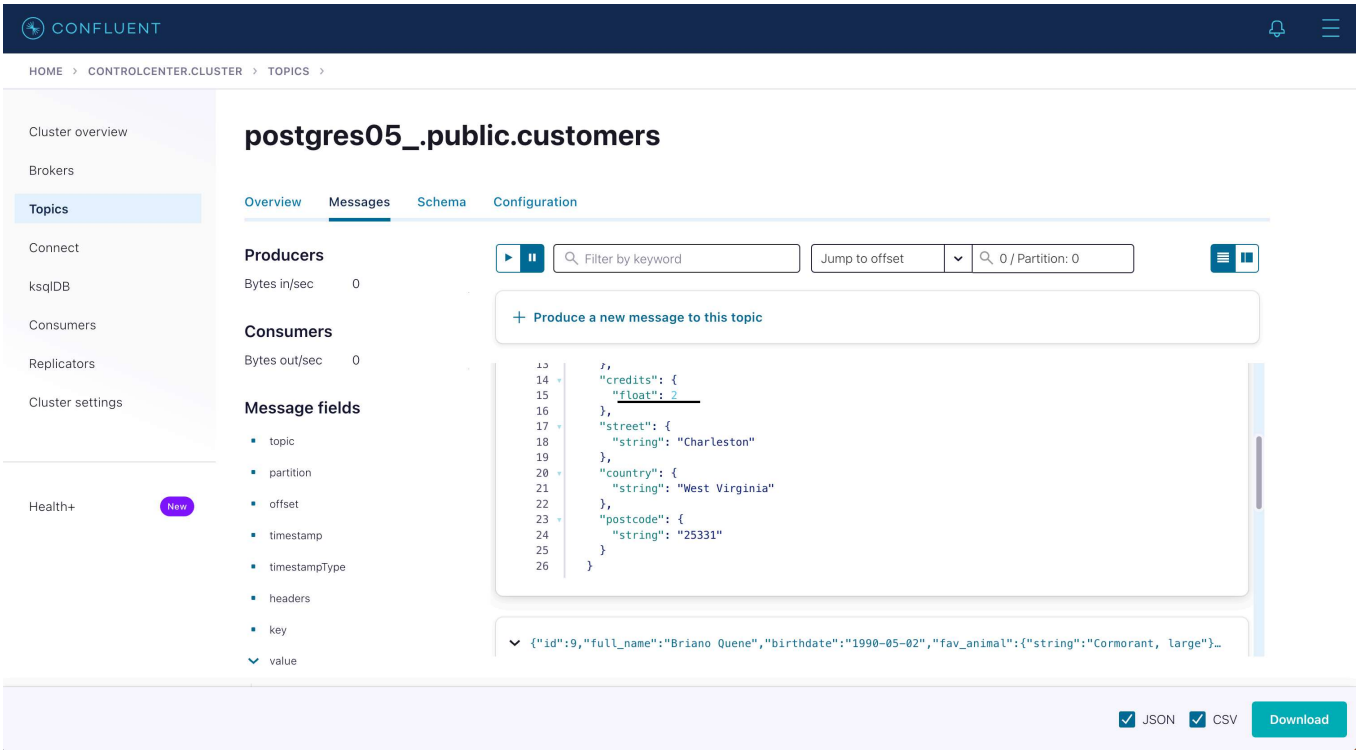
In this case, the order of the transforms is important:

```
curl -i -X PUT -H "Accept:application/json" \
-H  "Content-Type:application/json" http://localhost:8083/connectors/postgres-source-connector-customer05/config \
-d '{
                "connector.class": "io.debezium.connector.postgresql.PostgresConnector",
                "tasks.max": "1",
                "database.hostname": "db",
                "database.port": "5432",
                "database.user": "dc16",
                "database.password": "'${PASS}'",
                "database.dbname" : "postgres",
                "schema.include.list": "public",
                "table.include.list": "public.customers",
```

```
                "topic.prefix": "postgres05_",
                "plugin.name": "pgoutput",
                "transforms": "extract,createkey,extractKeyFromStruct,castTypes,filterAmount",
                "transforms.extract.type": "io.debezium.transforms.ExtractNewRecordState",
                "transforms.createkey.type": "org.apache.kafka.connect.transforms.ValueToKey",
                "transforms.createkey.fields": "id"  ,

"transforms.extractKeyFromStruct.type":"org.apache.kafka.connect.transforms.ExtractField$Key",
                "transforms.extractKeyFromStruct.field":"id",
                "transforms.filterAmount.type"              :
"io.confluent.connect.transforms.Filter$Value",
                "transforms.filterAmount.filter.condition": "$[?(@.credits < 42)]",
                "transforms.filterAmount.filter.type": "include",
                "transforms.castTypes.type"          :
"org.apache.kafka.connect.transforms.Cast$Value",
                "transforms.castTypes.spec"          : "credits:float32",
                "slot.name" : "05"
}'
```

Check in Confluent Control Center if the messages in the topic `postgres05.public.customers` are just the ones that credits field value is less than 42.



- Cast

Lab 9: Single Message Transforms - InsertField$Value

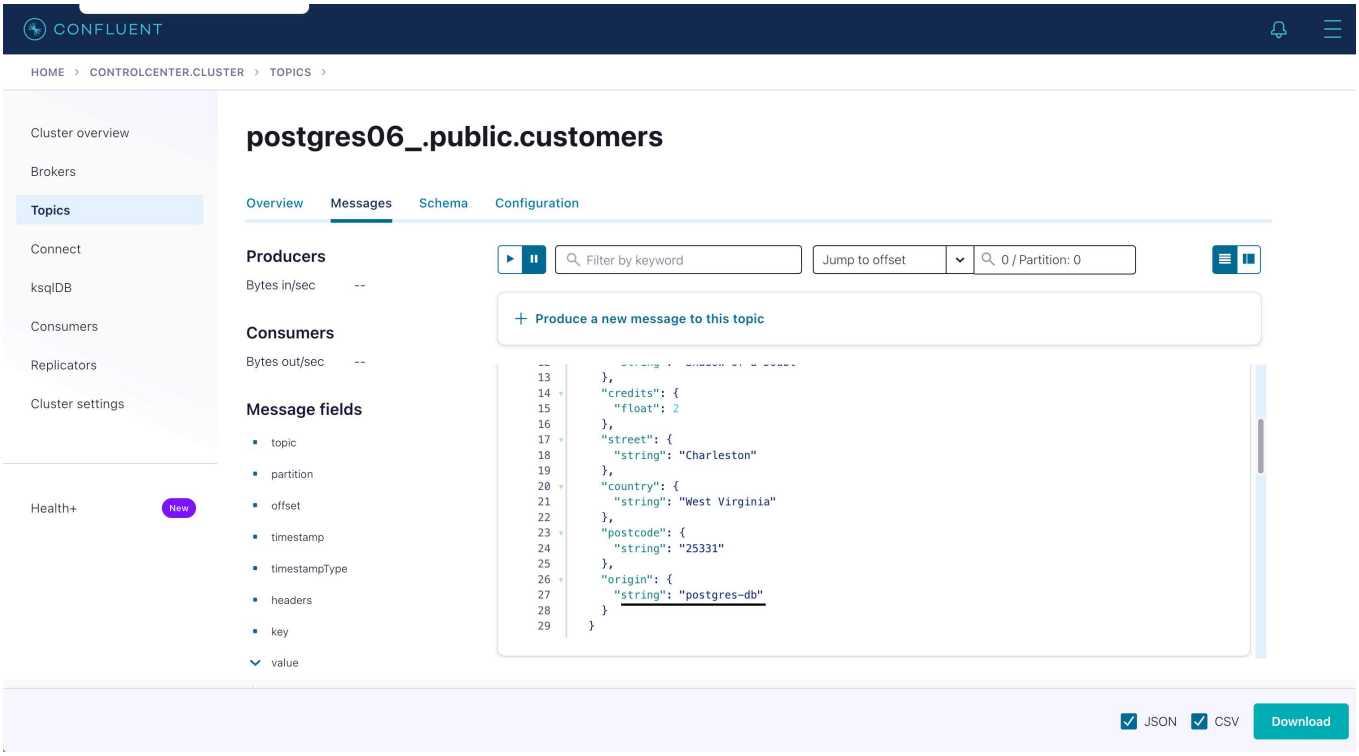
When ingesting data from a source (and there are several sources), it can be useful to add fields to store information such as the database from which it was read.

We can use SMT **InsertField$Value** for static values and add information in each message.

```
curl -i -X PUT -H "Accept:application/json" \
-H  "Content-Type:application/json" http://localhost:8083/connectors/postgres-source-
connector-customer06/config \
-d '{
                "connector.class": "io.debezium.connector.postgresql.PostgresConnector",
                "tasks.max": "1",
                "database.hostname": "db",
                "database.port": "5432",
                "database.user": "dc16",
                "database.password": "'${PASS}'",
                "database.dbname" : "postgres",
                "schema.include.list": "public",
                "table.include.list": "public.customers",
                "topic.prefix": "postgres06_",
                "plugin.name": "pgoutput",
                "transforms":
"extract,createkey,extractKeyFromStruct,insertStaticField1,castTypes",
                "transforms.extract.type": "io.debezium.transforms.ExtractNewRecordState",
                "transforms.createkey.type": "org.apache.kafka.connect.transforms.ValueToKey",
                "transforms.createkey.fields": "id"  ,

"transforms.extractKeyFromStruct.type":"org.apache.kafka.connect.transforms.ExtractField$Key",
                "transforms.extractKeyFromStruct.field":"id",
                "transforms.insertStaticField1.type"        :
"org.apache.kafka.connect.transforms.InsertField$Value",
                "transforms.insertStaticField1.static.field": "origin",
                "transforms.insertStaticField1.static.value": "postgres-db",
                "transforms.castTypes.type"          :
"org.apache.kafka.connect.transforms.Cast$Value",
                "transforms.castTypes.spec"          : "credits:float32",
                "slot.name" : "06"
}'
```

The resulting message that's written to Kafka includes the static data from the source system that is going to be useful to easily identify where the messages come from. Check in Confluent Control Center if the messages in the topic `postgres06.public.customers` have a new field `origin` with value `postgres-db`.



- Cast

Lab 10: Transforming data in realtime with ksqlDB

We now have all the data we need being streamed in realtime to Confluent Platform we can make some transformations before sending the data to MongoDB. We are going to use Confluent Center but if you prefer to use the KsqlDB CLI, execute the following command to access:

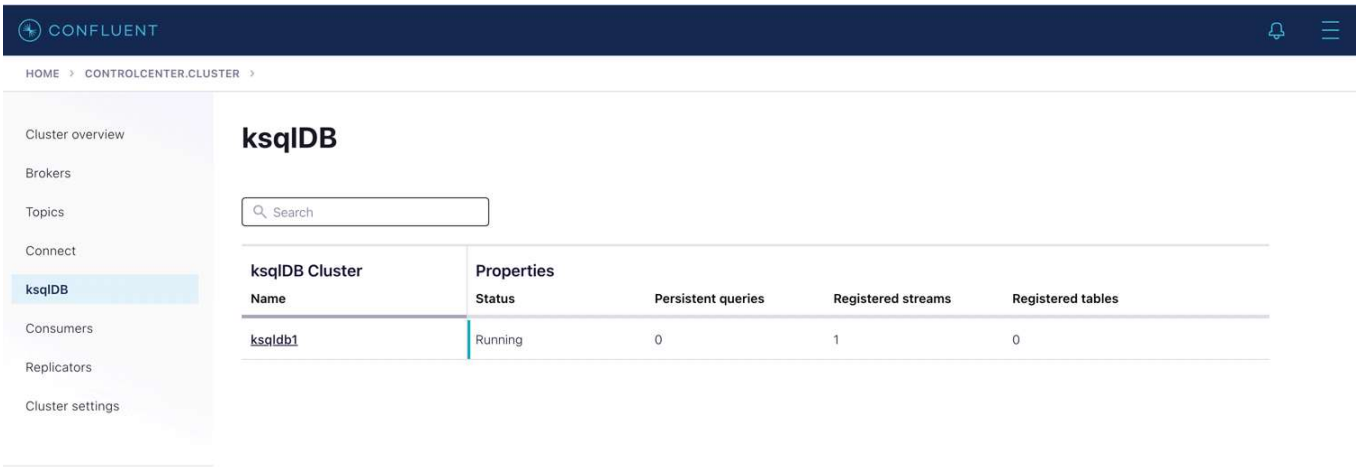
Start the ksqlDB CLI

To start the ksqlDB CLI run the following command:

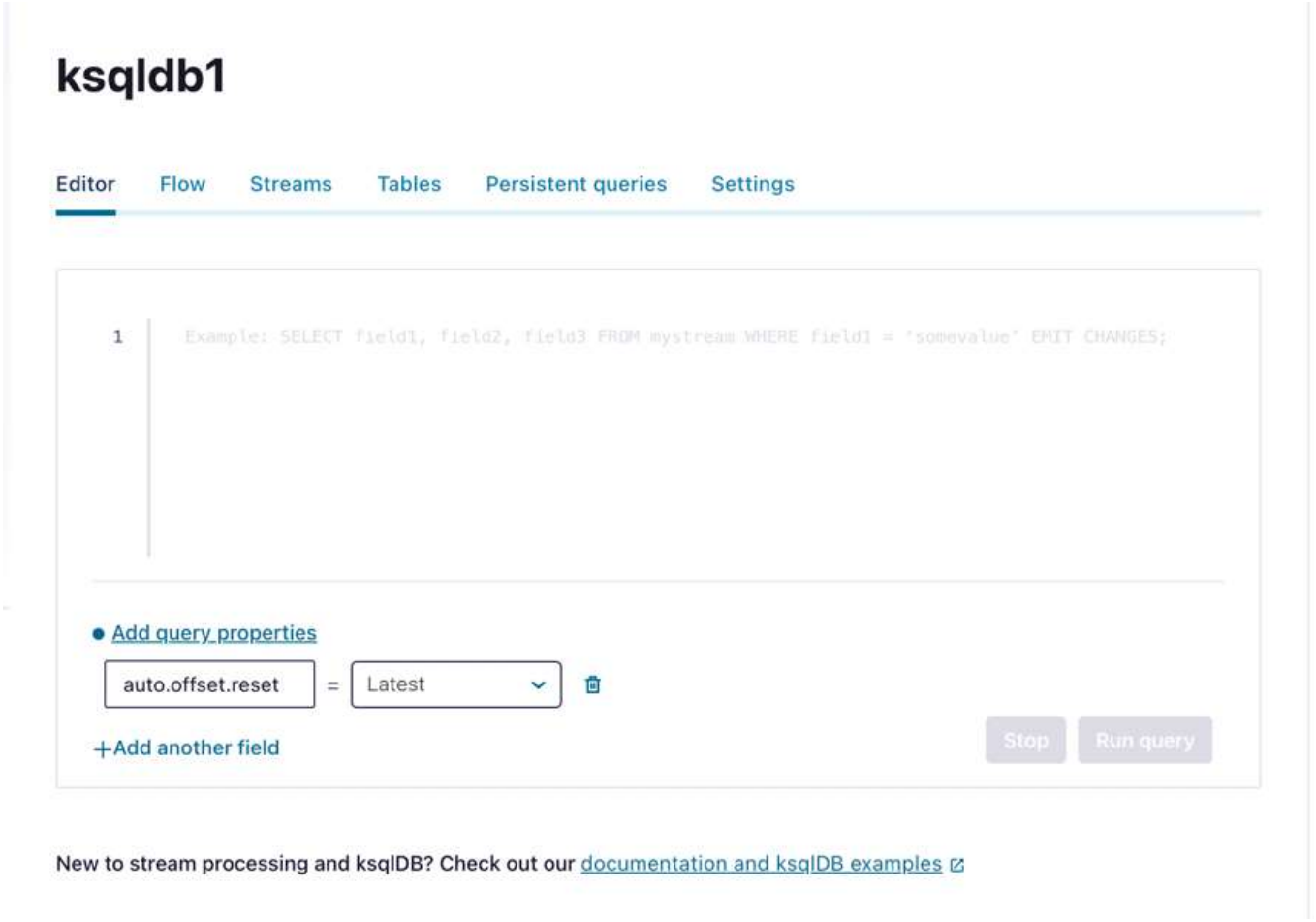
```
docker exec -it ksqldb-cli ksql http://ksqldb-server:8088
```

Start the ksqlDB in Control Center

Go to Confluent platform and select on the left hand side menu ksqlDB:



Once you are there, select the ksqbd1 cluster:



Now you are in the Confluent Platform UI ksqlDB Editor. Please select in the auto.offset.reset list Earliest:

We are going to create two streams, please copy them from the instructions below and create them using the KsqlDB editor.

```
CREATE STREAM customers WITH (KAFKA_TOPIC='postgres06_.public.customers',
PARTITIONS=1, VALUE_FORMAT='AVRO');

CREATE STREAM customers_struct AS SELECT
     ID  ,
     FULL_NAME ,
     BIRTHDATE ,
     FAV_ANIMAL ,
     FAV_COLOUR ,
     FAV_MOVIE ,
     CREDITS ,
     STRUCT(STREET:= STREET ,COUNTRY:=COUNTRY , POSTCODE:=POSTCODE)  ADDRESS,
     ORIGIN
FROM CUSTOMERS
PARTITION BY ID
EMIT CHANGES;
```

If you want to double check the data from the streams you just created you can execute the following queries in the ksqlDB Editor: (Remember the auto.offset.reset list Earliest)

```
SELECT * FROM customers EMIT CHANGES;
SELECT * FROM customers_struct EMIT CHANGES;
```

Further Reading

- ksqlDB Overview
- ksqlDB Streams

Lab 11: Sink data from Confluent to MongoDB

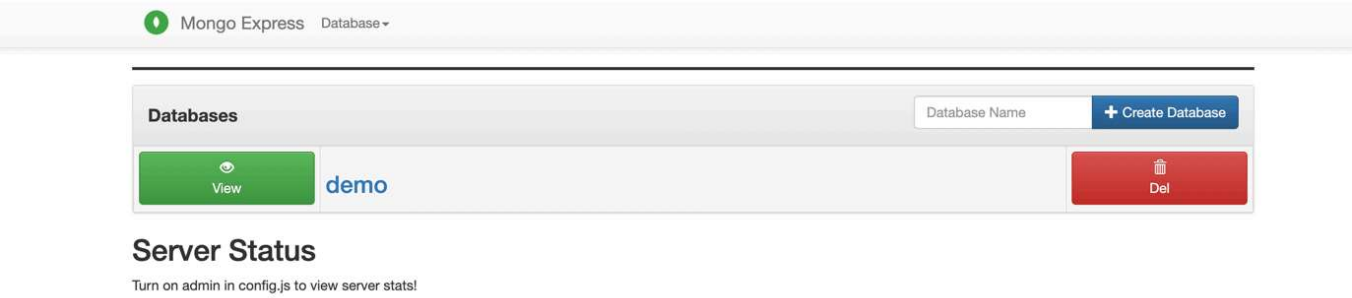
We already have the data processed available in Confluent. To make the data available in **MongoDB** database we are going to leverage the complete connect portfolio that is already built as we did in previous steps. But in this case the connector used is going to be the **MongoDB Sink Connector .**

To start migrating the data from **CUSTOMERS_STRUCT** stream we created in the previous step, we will need to execute the following cURL command:

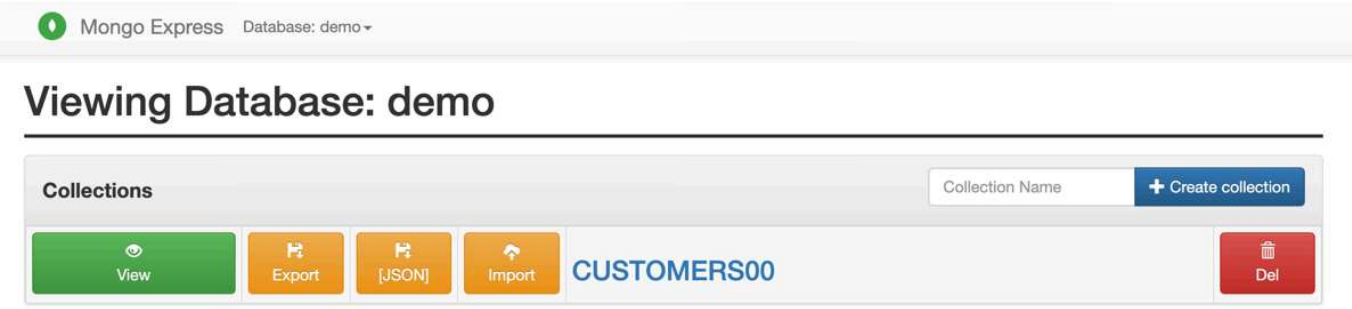
```
curl -i -X PUT -H "Accept:application/json" \
-H  "Content-Type:application/json" http://localhost:8083/connectors/mongodb-sink-connector-customer00/config \
-d '{
                "connector.class": "com.mongodb.kafka.connect.MongoSinkConnector",
                "topics": "CUSTOMERS_STRUCT",
                "tasks.max": "1",
                "connection.uri": "mongodb://user:pass@mymongodb:27017/?authSource=demo",
                "key.converter": "org.apache.kafka.connect.storage.StringConverter",
                "value.converter": "io.confluent.connect.avro.AvroConverter",
                "value.converter.schema.registry.url": "http://schema-registry:8081",
                "value.converter.schemas.enable": true,
                "database": "demo",
                "collection": "CUSTOMERS00"
     }'
```

Once it has been executed and created we can go and check to MongoDB if the data has arrived accessing to the following url:

MongoDB UI



Accessing to demo database and we will see the first Collection called CUSTOMERS00 that has been created by the connector:



If we inspect the data we will see something like this (Please look at the first column (_id), we'll cover that in a minute):

_id	FULL_NAME	BIRTHDATE	FAV_ANIMAL	FAV_COLOUR	FAV_MOVIE	CREDITS	ADDRESS
644a2ae52b707a6682ed6753	Leone Puxley	1995-02-06	Violet-eared waxbill	Puce	Oh! What a Lovely War	53.4900016784668	{"STREET": "Lynchburg", "COUNTRY": "Virginia", "POSTCODE": "24515"}

Further Reading

- link:https://www.mongodb.com/docs/kafka-connector/current/sink-connector/configuration-properties/all-properties/ [MongoDB Sink Connector]

Lab 12: MongoDB Sink Connector Configurations and Single Message Transforms - document.id.strategy and HoistField$Key

MongoDB is a document database and the **_id** is the document identifier. If we don't provide one in the connector configuration one will be created as we can see in the picture from the previous step. But that identifier does not mean anything, if an update happens it is not going to replace the data in the existing document, Mongo will create another document for the update with the new data with new **_id**.

To check that go to PgAdmin UI and update one row:

```
UPDATE public.customers
SET fav_animal = 'Mouse'
 WHERE id = 1;

SELECT id, full_name, birthdate, fav_animal, fav_colour, fav_movie
    FROM public.customers;
```

We will have both documents in MongoDB with different **_id** values, that's because they don't have a proper document identifier, it has been created randomly.

_id	FULL_NAME	BIRTHDATE	FAV_ANIMAL	FAV_COLOUR	FAV_MOVIE	CREDITS	ADDRESS
644a2ae52b707a6682ed6753	Leone Puxley	1995-02-06	Violet-eared waxbill	Puce	Oh! What a Lovely War	53.4900016784668	{"STREET": "Lynchburg", "COUNTRY": "Virginia", "POSTCODE": "24515"}

MongoDB Sink connector has configurations to solve the problem depending on the approach that you need. In this case we want to use the Kafka message key as we already have a proper identifier there. Using the configuration **document.id.strategy** and kafka connect transform **HoistField$Key**, you will achieve that:

```
curl -i -X PUT -H "Accept:application/json" \
-H  "Content-Type:application/json" http://localhost:8083/connectors/mongodb-sink-connector-customer01/config \
-d '{

                "connector.class": "com.mongodb.kafka.connect.MongoSinkConnector",
                "topics": "CUSTOMERS_STRUCT",
                "tasks.max": "1",
                "connection.uri": "mongodb://user:pass@mymongodb:27017/?authSource=demo",
                "key.converter": "org.apache.kafka.connect.converters.IntegerConverter",
                "key.converter.schemas.enable": false,
                "value.converter": "io.confluent.connect.avro.AvroConverter",
                "value.converter.schema.registry.url": "http://schema-registry:8081",
                "value.converter.schemas.enable": true,
                "database": "demo",
                "collection": "CUSTOMERS01",
                "document.id.strategy": "com.mongodb.kafka.connect.sink.processor.id.strategy.ProvidedInKeyStrategy",
                "transforms": "hk",
                "transforms.hk.type": "org.apache.kafka.connect.transforms.HoistField$Key",
                "transforms.hk.field": "_id"
    }'
```

After creating the connector we will have another collection **CUSTOMERS01**, check if it has the updated data you did before in the document _id=1 and there is no other document for that data.

Further Reading

- Hoistfield
- Mongo Sink Connector - id Strategy

Lab 13: MongoDB Sink Connector Configurations and Single Message Transforms - Flatten$Value

As we can observe from the previous collections, we have the address data in a struct that comes from the stream:

We can flatten that data using the SMT **Flatten$Value** if we need to have each field inside the struct in the first level field in the document.

```
curl -i -X PUT -H "Accept:application/json" \
-H  "Content-Type:application/json" http://localhost:8083/connectors/mongodb-sink-connector-customer02/config \
-d '{

                "connector.class": "com.mongodb.kafka.connect.MongoSinkConnector",
                "topics": "CUSTOMERS_STRUCT",
                "tasks.max": "1",
                "connection.uri": "mongodb://user:pass@mymongodb:27017/?authSource=demo",
                "key.converter": "org.apache.kafka.connect.converters.IntegerConverter",
                "key.converter.schemas.enable": false,
                "value.converter": "io.confluent.connect.avro.AvroConverter",
                "value.converter.schema.registry.url": "http://schema-
```

```
registry:8081",
                "value.converter.schemas.enable": true,
                "database": "demo",
                "collection": "CUSTOMERS02",
                "document.id.strategy":
"com.mongodb.kafka.connect.sink.processor.id.strategy.ProvidedInKeyStrategy",
                "transforms": "hk,flatten",
                "transforms.hk.type":
"org.apache.kafka.connect.transforms.HoistField$Key",
                "transforms.hk.field": "_id",
                "transforms.flatten.type"       :
"org.apache.kafka.connect.transforms.Flatten$Value",
                "transforms.flatten.delimiter"  : "_"
    }'
```

After creating the connector we will have another collection **CUSTOMERS02**, check if it has the address data flatten.

ADDRESS_STREET	ADDRESS_COUNTRY	ADDRESS_POSTCODE
Lynchburg	Virginia	24515

Further Reading

- Flatten
- Mongo Sink Connector - id Strategy

Lab 14: MongoDB Sink Connector Configurations and Single Message Transforms - writemodel.strategy and UpdateOneTimestampsStrategy

MongoDB also has configurations to insert columns related to the timestamps about **the insertion and updates** on the data. We can use **writemodel.strategy** to achieve that, using **UpdateOneTimestampsStrategy** that is going to add fields with the exact info **about the timestamp when the document was inserted and updated in MongoDB.**

```
curl -i -X PUT -H "Accept:application/json" \
-H  "Content-Type:application/json" http://localhost:8083/connectors/mongodb-sink-
connector-customer03/config \
-d '{
                "connector.class": "com.mongodb.kafka.connect.MongoSinkConnector",
                "topics": "CUSTOMERS_STRUCT",
                "tasks.max": "1",
```

```
                "connection.uri": "mongodb://user:pass@mymongodb:27017/?
authSource=demo",
                "key.converter":
"org.apache.kafka.connect.converters.IntegerConverter",
                "key.converter.schemas.enable": false,
                "value.converter": "io.confluent.connect.avro.AvroConverter",
                "value.converter.schema.registry.url": "http://schema-
registry:8081",
                "value.converter.schemas.enable": true,
                "database": "demo",
                "collection": "CUSTOMERS03",
                "document.id.strategy":
"com.mongodb.kafka.connect.sink.processor.id.strategy.ProvidedInKeyStrategy",
                "writemodel.strategy":
"com.mongodb.kafka.connect.sink.writemodel.strategy.UpdateOneTimestampsStrategy",
                "transforms": "hk,flatten",
                "transforms.hk.type":
"org.apache.kafka.connect.transforms.HoistField$Key",
                "transforms.hk.field": "_id",
                "transforms.flatten.type"       :
"org.apache.kafka.connect.transforms.Flatten$Value",
                "transforms.flatten.delimiter"  : "_"
 }'
```

After creating the connector we will have another collection **CUSTOMERS03**, check if it has two new columns **_insertedTS** and **_modifiedTS.** As this new collection has been just created both values will be the same. We can update data in Postgres database in order to see the different timestamps in both columns:

```
UPDATE public.customers
SET fav_animal = 'Mickey Mouse'
 WHERE id = 1;
```

After updating the data we will see different values betweeen both columns:

_insertedTS	_modifiedTS
Thu Apr 27 2023 08:48:53 GMT+0000 (Coordinated Universal Time)	Thu Apr 27 2023 08:54:47 GMT+0000 (Coordinated Universal Time)

Further Reading

- Mongo Sink Connector - Write Stragegies

Lab 15: MongoDB Sink Connector Configurations and Single Message Transforms - MaskField$Value

Maybe sensitive data exists that we don't want that the downstreams know. We can mask that information using SMT **MaskField$Value**. We are going to mask the address information about our customers.

```
curl -i -X PUT -H "Accept:application/json" \
-H  "Content-Type:application/json" http://localhost:8083/connectors/mongodb-sink-
connector-customer04/config \
-d '{            "connector.class": "com.mongodb.kafka.connect.MongoSinkConnector",
               "topics": "CUSTOMERS_STRUCT",
               "tasks.max": "1",
               "connection.uri": "mongodb://user:pass@mymongodb:27017/?
authSource=demo",
               "key.converter":
"org.apache.kafka.connect.converters.IntegerConverter",
               "key.converter.schemas.enable": false,
               "value.converter": "io.confluent.connect.avro.AvroConverter",
               "value.converter.schema.registry.url": "http://schema-
registry:8081",
               "value.converter.schemas.enable": true,
               "database": "demo",
               "collection": "CUSTOMERS04",
               "document.id.strategy":
"com.mongodb.kafka.connect.sink.processor.id.strategy.ProvidedInKeyStrategy",
               "writemodel.strategy":
"com.mongodb.kafka.connect.sink.writemodel.strategy.UpdateOneTimestampsStrategy",
               "transforms": "hk,flatten,maskAddress",
               "transforms.hk.type":
"org.apache.kafka.connect.transforms.HoistField$Key",
               "transforms.hk.field": "_id",
               "transforms.flatten.type"       :
"org.apache.kafka.connect.transforms.Flatten$Value",
               "transforms.flatten.delimiter"  : "_",
               "transforms.maskAddress.type"        :
"org.apache.kafka.connect.transforms.MaskField$Value",
               "transforms.maskAddress.fields"      :
"ADDRESS_COUNTRY,ADDRESS_POSTCODE,ADDRESS_STREET",
```

```
                    "transforms.maskAddress.replacement" : "XXXXXXXXXX"
}'
```

After creating the connector we will have another collection **CUSTOMERS04**, check if it has the flattened address data has been masked:

Further Reading

- Maskfield

Lab 16: MongoDB Sink Connector Configurations and Single Message Transforms - InsertField$Value

We can also add metadata from kafka in case we need to use it in your consumer application. It can be achieved using SMT **InsertField$Value.** Let's add the topic, partition and offset.

```
curl -i -X PUT -H "Accept:application/json" \
-H  "Content-Type:application/json" http://localhost:8083/connectors/mongodb-sink-connector-customer05/config \
-d '{
                "connector.class": "com.mongodb.kafka.connect.MongoSinkConnector",
                "topics": "CUSTOMERS_STRUCT",
                "tasks.max": "1",
                "connection.uri": "mongodb://user:pass@mymongodb:27017/?authSource=demo",
                "key.converter": "org.apache.kafka.connect.converters.IntegerConverter",
                "key.converter.schemas.enable": false,
                "value.converter": "io.confluent.connect.avro.AvroConverter",
                "value.converter.schema.registry.url": "http://schema-registry:8081",
                "value.converter.schemas.enable": true,
                "database": "demo",
                "collection": "CUSTOMERS05",
                "document.id.strategy":
"com.mongodb.kafka.connect.sink.processor.id.strategy.ProvidedInKeyStrategy",
                "writemodel.strategy":
"com.mongodb.kafka.connect.sink.writemodel.strategy.UpdateOneTimestampsStrategy",
                "transforms": "hk,insertPartition,insertOffset,insertTopic",
```

```
                "transforms.hk.type": "org.apache.kafka.connect.transforms.HoistField$Key",
                "transforms.hk.field": "_id",
                "transforms.insertPartition.type"           :
"org.apache.kafka.connect.transforms.InsertField$Value",
                "transforms.insertPartition.partition.field": "kafkaPartition",
                "transforms.insertOffset.type"              :
"org.apache.kafka.connect.transforms.InsertField$Value",
                "transforms.insertOffset.offset.field"      : "kafkaOffset",
                "transforms.insertTopic.type"               :
"org.apache.kafka.connect.transforms.InsertField$Value",
                "transforms.insertTopic.topic.field"        : "kafkaTopic"
    }'
```

After creating the connector we will have another collection **CUSTOMERS05**, check if it has three new columns `kafkaPartition`, `kafkaOffset` and `kafkaTopic`:

kafkaOffset	kafkaPartition	kafkaTopic
11	0	CUSTOMERS_STRUCT

Further Reading

- Maskfield

Lab 17: MongoDB Sink Connector Configurations and Single Message Transforms - post.processor.chain and KafkaMetaAdder

But mongoDB sink connector also has a post processor configuration to achieve the same that we did in the previous lab: **post.processor.chain** configuration with the value **KafkaMetaAdder.**

```
curl -i -X PUT -H "Accept:application/json" \
-H  "Content-Type:application/json" http://localhost:8083/connectors/mongodb-sink-connector-
customer06/config \
-d '{
                "connector.class": "com.mongodb.kafka.connect.MongoSinkConnector",
                "topics": "CUSTOMERS_STRUCT",
                "tasks.max": "1",
                "connection.uri": "mongodb://user:pass@mymongodb:27017/?authSource=demo",
                "key.converter": "org.apache.kafka.connect.converters.IntegerConverter",
```

```
                "key.converter.schemas.enable": false,
                "value.converter": "io.confluent.connect.avro.AvroConverter",
                "value.converter.schema.registry.url": "http://schema-registry:8081",
                "value.converter.schemas.enable": true,
                "database": "demo",
                "collection": "CUSTOMERS06",
                "document.id.strategy":
"com.mongodb.kafka.connect.sink.processor.id.strategy.ProvidedInKeyStrategy",
                "writemodel.strategy":
"com.mongodb.kafka.connect.sink.writemodel.strategy.UpdateOneTimestampsStrategy",
                "post.processor.chain": "com.mongodb.kafka.connect.sink.processor.KafkaMetaAdder",
                "transforms": "hk",
                "transforms.hk.type": "org.apache.kafka.connect.transforms.HoistField$Key",
                "transforms.hk.field": "_id"
    }'
```

After creating the connector we will have another collection **CUSTOMERS06**, check if it has a new columns `topic-partition-offset` with the metadata in its value:

Further Reading

- MongoDB Sink Connector - post-processors

Lab 18: MongoDB Sink Connector Configurations and Single Message Transforms - post.processor.chain and BlockListValueProjector

The post processors also let us select or avoid the data that we want to make available or not for the downstream. We are going to use **BlockListValueProjector** to not send address information to **MongoDB.**

```
curl -i -X PUT -H "Accept:application/json" \
-H  "Content-Type:application/json" http://localhost:8083/connectors/mongodb-sink-connector-
customer07/config \
-d '{
```

```
                "connector.class": "com.mongodb.kafka.connect.MongoSinkConnector",
                "topics": "CUSTOMERS_STRUCT",
                "tasks.max": "1",
                "connection.uri": "mongodb://user:pass@mymongodb:27017/?authSource=demo",
                "key.converter": "org.apache.kafka.connect.converters.IntegerConverter",
                "key.converter.schemas.enable": false,
                "value.converter": "io.confluent.connect.avro.AvroConverter",
                "value.converter.schema.registry.url": "http://schema-registry:8081",
                "value.converter.schemas.enable": true,
                "database": "demo",
                "collection": "CUSTOMERS07",
                "document.id.strategy":
"com.mongodb.kafka.connect.sink.processor.id.strategy.ProvidedInKeyStrategy",
                "writemodel.strategy":
"com.mongodb.kafka.connect.sink.writemodel.strategy.UpdateOneTimestampsStrategy",
                "post.processor.chain":
"com.mongodb.kafka.connect.sink.processor.BlockListValueProjector",
                "value.projection.type": "BlockList",
                "value.projection.list": "ADDRESS",
                "transforms": "hk",
                "transforms.hk.type": "org.apache.kafka.connect.transforms.HoistField$Key",
                "transforms.hk.field": "_id"
        }'
```

After creating the connector we will have another collection **CUSTOMERS07**, check if ithe address data has disappeared:

_id	BIRTHDATE	CREDITS	FAV_ANIMAL	FAV_COLOUR	FAV_MOVIE	FULL_NAME	ORIGIN	_insertedTS	_modifiedTS
1	1995-02-06	53.4900016784668	Mickey Mouse	Puce	Oh! What a Lovely War	Leone Puxley	postgres-db	Thu Apr 27 2023 09:10:58 GMT+0000 (Coordinated Universal Time)	Thu Apr 27 2023 09:10:58 GMT+0000 (Coordinated Universal Time)

Further Reading

- MongoDB Sink Connector - post-processors

Lab 19: Error Handing - Dead Letter Queue

If you want to avoid that an error makes a connector to be in a failed state you have the dql handling errors option. An invalid record may occur for a number of reasons. For Connect, errors that may occur are typically serialization and deserialization (serde) errors. For example, an error occurs when a record arrives at the sink connector in JSON format, but the sink connector configuration is expecting another format, like Avro. Using error handling with DLQ, the connector does not stop when serde errors occur. Instead, the connector continues processing records and sends the errors to a Dead Letter Queue (DLQ). You can use the record headers in a DLQ topic record to identify and troubleshoot an error when it occurs. Typically, these are configuration errors that can be easily corrected.

The following cURL command is prepared to have errors and sending them to the DLQ topic called `dlq_sink_08`:

```
curl -i -X PUT -H "Accept:application/json" \
-H  "Content-Type:application/json" http://localhost:8083/connectors/mongodb-sink-connector-customer08/config \
-d '{          "connector.class": "com.mongodb.kafka.connect.MongoSinkConnector",
               "topics": "postgres06_.public.customers",
               "tasks.max": "1",
               "connection.uri": "mongodb://user:pass@mymongodb:27017/?authSource=demo",
               "key.converter": "org.apache.kafka.connect.converters.IntegerConverter",
               "key.converter.schemas.enable": true,
               "value.converter": "io.confluent.connect.avro.AvroConverter",
               "value.converter.schema.registry.url": "http://schema-registry:8081",
               "value.converter.schemas.enable": true,
               "database": "demo",
               "collection": "CUSTOMERS08",
               "document.id.strategy":
"com.mongodb.kafka.connect.sink.processor.id.strategy.ProvidedInKeyStrategy",
               "transforms": "hk",
               "transforms.hk.type": "org.apache.kafka.connect.transforms.HoistField$Key",
               "transforms.hk.field": "_id",
               "errors.tolerance" : "all",
               "mongo.errors.tolerance": "all",
               "errors.deadletterqueue.topic.replication.factor" : 1,
               "errors.deadletterqueue.topic.name" : "dlq_sink_08",
               "errors.deadletterqueue.context.headers.enable": true
     }'
```

After creating the connector the topic `dlq_sink_08` will be created in our Confluent Platform Cluster, we can go to the topics menu and check the messages that have failed there:

```
  {
    "key": "__connect.errors.stage",
    "stringValue": "KEY_CONVERTER"
  },
  {
    "key": "__connect.errors.class.name",
    "stringValue": "org.apache.kafka.connect.converters.IntegerConverter"
  },
  {
    "key": "__connect.errors.exception.class.name",
    "stringValue": "org.apache.kafka.connect.errors.DataException"
  },
  {
    "key": "__connect.errors.exception.message",
    "stringValue": "Failed to deserialize integer: "
  },
  {
    "key": "__connect.errors.exception.stacktrace",
    "stringValue": "org.apache.kafka.connect.errors.DataException: Failed to deserialize
integer: \n\tat
org.apache.kafka.connect.converters.NumberConverter.toConnectData(NumberConverter.java:109)\n\t
at
org.apache.kafka.connect.converters.IntegerConverter.toConnectData(IntegerConverter.java:32)\n\
tat org.apache.kafka.connect.storage.Converter.toConnectData(Converter.java:88)\n\tat
org.apache.kafka.connect.runtime.WorkerSinkTask.lambda$convertAndTransformRecord$3(WorkerSinkTa
sk.java:513)\n\tat
org.apache.kafka.connect.runtime.errors.RetryWithToleranceOperator.execAndRetry(RetryWithTolera
nceOperator.java:173)\n\tat
org.apache.kafka.connect.runtime.errors.RetryWithToleranceOperator.execAndHandleError(RetryWith
ToleranceOperator.java:207)\n\tat
org.apache.kafka.connect.runtime.errors.RetryWithToleranceOperator.execute(RetryWithToleranceOp
erator.java:149)\n\tat
org.apache.kafka.connect.runtime.WorkerSinkTask.convertAndTransformRecord(WorkerSinkTask.java:5
13)\n\tat
org.apache.kafka.connect.runtime.WorkerSinkTask.convertMessages(WorkerSinkTask.java:493)\n\tat
org.apache.kafka.connect.runtime.WorkerSinkTask.poll(WorkerSinkTask.java:332)\n\tat
org.apache.kafka.connect.runtime.WorkerSinkTask.iteration(WorkerSinkTask.java:234)\n\tat
org.apache.kafka.connect.runtime.WorkerSinkTask.execute(WorkerSinkTask.java:203)\n\tat
```


Further Reading

- MongoDB Sink Connector - post-processors

Wrapping up

During this workshop we have seen how the Confluent Platform and Confluent Cloud can be used to build event driven, real time applications that span the data center and public cloud.