Confluent Developer Skills for Building Apache Kafka®

Exercise Book

Version 7.0.0-v1.0.5

Table of Contents

Copyright & Trademarks . . . 1

Lab 01 Fundamentals of Apache Kafka . . . 2

a. Introduction . . . 2

b. Using Kafka's Command-Line Tools . . . 11

Lab 02 Producing Messages to Kafka . . . 18

a. Kafka Producer (Java, C#, Python) . . . 18

Lab 04 Consuming Messages from Kafka . . . 26

a. Kafka Consumer (Java, C#, Python) . . . 26

Lab 07 Schema Management in Apache Kafka . . . 32

a. Schema Registry, Avro Producer and Consumer (Java, C#, Python) . . . 32

Lab 08 Stream Processing with Kafka Streams . . . 40

a. Kafka Streams (Java) . . . 40

Lab 09 Event Streaming Apps with ksqlDB . . . 45

a. ksqlDB - Join a Stream and a Table . . . 45

Lab 11 Data Pipelines with Kafka Connect . . . 52

a. Kafka Connect - Database to Kafka . . . 52

Lab 12 Challenges with Offsets . . . 64

a. Kafka Consumer - offsetsForTimes (Java, C#, Python) . . . 64

Lab 13 Partitioning Considerations . . . 69

a. Increasing the Topic Partitions . . . 69

Appendix A: Running All Labs with Docker . . . 74

Running Labs in Docker for Desktop . . . 74

Copyright & Trademarks

Lab 01 Fundamentals of Apache Kafka

a. Introduction

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

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

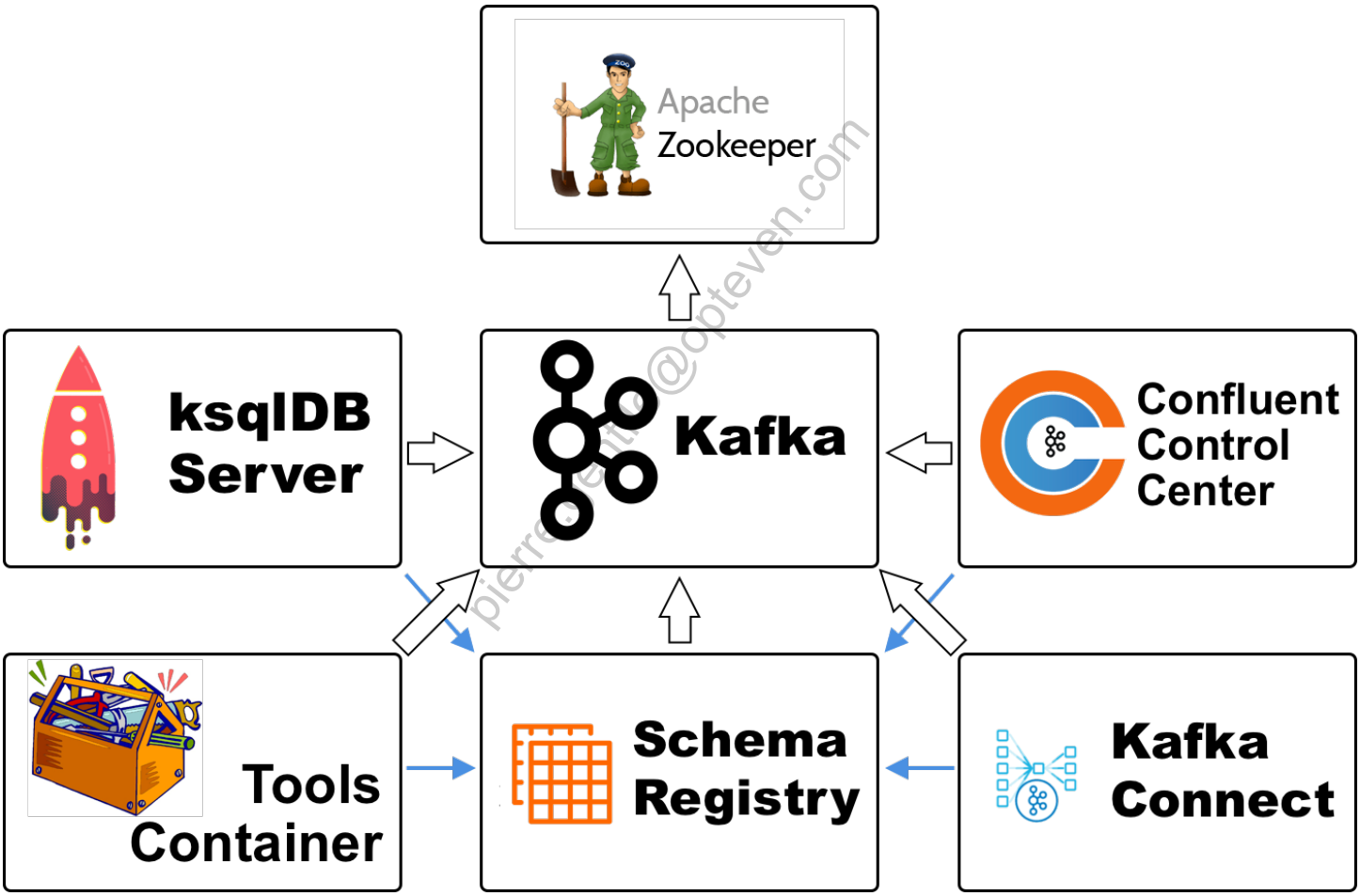


Table 1. Components of the Confluent Platform

Alias	Description
zookeeper	ZooKeeper
kafka	Kafka Broker
schema-registry	Schema Registry

Alias	Description
connect	Kafka Connect
ksqldb-server	ksqlDB Server
control-center	Confluent Control Center
tools	secondary location for tools run against the cluster

As you progress through the exercises you will selectively turn on parts of your cluster as they are needed.

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

In this course we are using Confluent Platform version 7.0.0 which includes Kafka 3.0.0.

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

Alternative Lab Environments

As an alternative you can also download the VM to your laptop and run it in VirtualBox. Make sure you have the newest version of VirtualBox installed. Download the VM from this link:

- https://s3.amazonaws.com/confluent-training-images-us-east-1/training-ubuntu-20-04-jan2022.ova

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

Command Line Examples

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

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

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

Preparing the Labs

Welcome to your lab environment! You are connected as user **training**, password **training**.

If you haven't already done so, you should open the **Exercise Guide** that is located on the lab virtual machine. To do so, open the **Confluent Training Exercises** folder that is located on the lab virtual machine desktop. Then double-click the shortcut that is in the folder to open the **Exercise Guide**.

Copy and paste works best if you copy from the Exercise Guide on your lab virtual machine.

- Standard Ubuntu keyboard shortcuts will work: `Ctrl+C` → Copy, `Ctrl+V` → Paste
- In a Terminal window: `Ctrl+Shift+C` → Copy, `Ctrl+Shift+V` → Paste.

If you find these keyboard shortcuts are not working you can use the right-click context menu for copy and paste.

1. Open a terminal window
2. Clone the source code repository to the folder `confluent-dev` in your **home** directory:

```
$ cd ~
$ git clone --depth 1 --branch 7.0.0-v1.0.5 \
    https://github.com/confluentinc/training-developer-src.git \
    confluent-dev
```

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

3. Navigate to the `confluent-dev` folder:

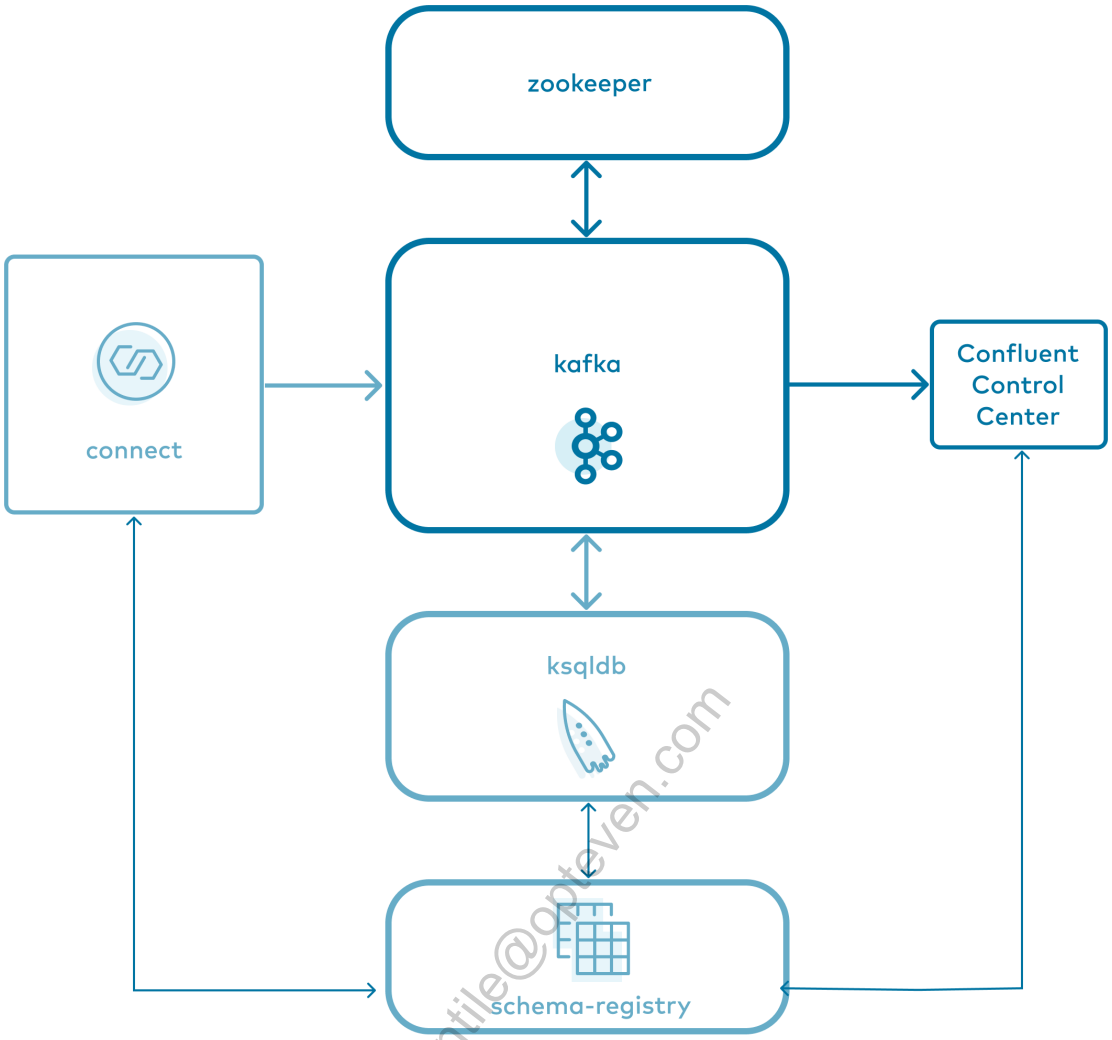
```
$ cd ~/confluent-dev
```

4. Start the Kafka cluster:

```
$ docker-compose up -d zookeeper kafka control-center
```

You should see something similar to this:

```
Creating network "confluent-dev_default" with the default driver
Creating control-center ... done
Creating kafka          ... done
Creating zookeeper      ... done
```

In the first steps of each exercise, you launch the containers needed for the exercise with `docker-compose up`.

If at any time you want to get your environment back to a clean state use `docker-compose down` to end all of your containers. Then return to your last `docker-compose up` to get back to the beginning of an exercise.

Exercises do not need to be completed in order. You can start from the beginning of any exercise at any time.

If you want to completely clear out your docker environment use the script on the VM at `~/docker-nuke.sh`. The nuke script will forcefully end all of your running docker containers.

5. Monitor the cluster with:

```
$ docker-compose ps

     Name                   Command            State
Ports
-------------------------------------------------------------------
--------------------------
control-center   /etc/confluent/docker/run   Up      0.0.0.0:9021-
>9021/tcp
kafka            /etc/confluent/docker/run   Up      0.0.0.0:9092-
>9092/tcp
zookeeper        /etc/confluent/docker/run   Up      0.0.0.0:2181-
>2181/tcp, 2888/tcp, 3888/tcp
```

All services should have `State` equal to `Up`.

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

```
$ docker stats

CONTAINER ID        NAME                CPU %         MEM USAGE /
LIMIT    MEM %    ...
e174ec2aaa51        zookeeper           0.00%         86.88MiB /
7.787GiB  1.09%
2bfac54019a2        kafka               0.01%         450.9MiB /
7.787GiB  5.65%
14c813cf0cf1        control-center      0.01%         376.7MiB /
7.787GiB  4.72%
```

Press `Ctrl+C` to exit the Docker statistics.

Testing the Installation

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

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

WATCHER::

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

OPTIONAL: Analyzing the Docker Compose File

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

```
KAFKA_ZOOKEEPER_CONNECT: zookeeper:2181
```

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

```
KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 1
```

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

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

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

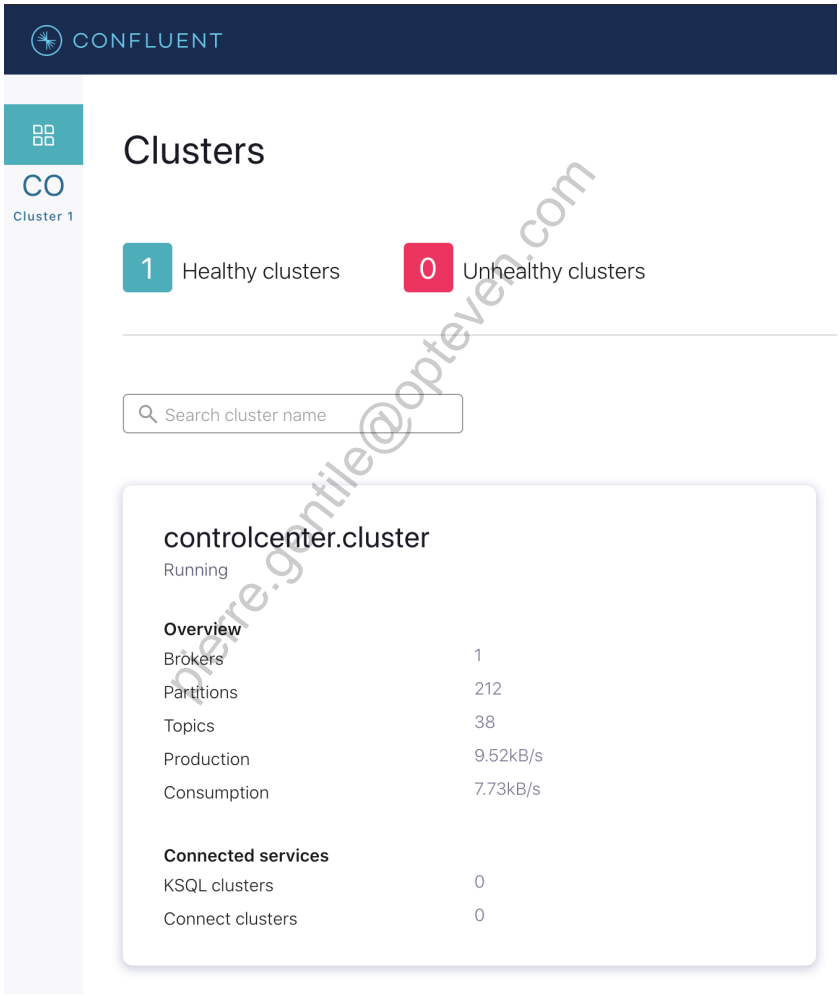
```
..._BOOTSTRAP_SERVERS: kafka:9092
```

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

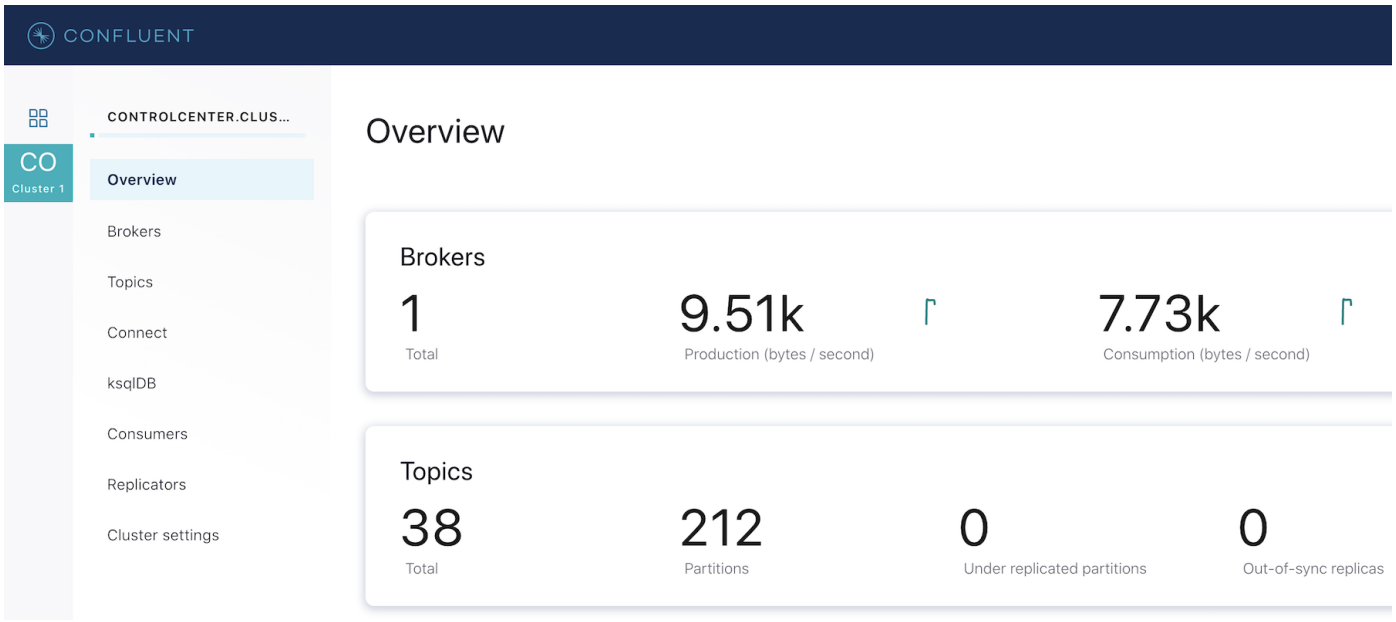
```
io.confluent.monitoring.clients.interceptor.MonitoringProducerInte
rceptor
io.confluent.monitoring.clients.interceptor.MonitoringConsumerInte
rceptor
```

Using Confluent Control Center

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



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



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

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

b. Using Kafka's Command-Line Tools

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

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

Prerequisites

1. Navigate to the `confluent-dev` folder:

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

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

```
$ docker-compose up -d zookeeper kafka control-center
```

If your containers are running from the previous exercise this command will simply tell you each container is up-to-date.

Console Producing and Consuming

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

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

```
$ kafka-topics
```

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

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

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

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

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

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

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

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

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

```
$ kafka-console-producer --bootstrap-server kafka:9092 --topic
testing
```

The tool prompts you with a `>`.

5. At this prompt type:

```
> some data
```

And click **Enter**.

6. Now type:

```
> more data
```

And click **Enter**.

7. Type:

```
> final data
```

And click **Enter**.

8. Now we will use a Consumer to retrieve the data that was produced. Open a new terminal window and run the command:


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

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

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

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

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

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

10. Press `Ctrl+D` to exit the `kafka-console-producer` program.
11. Press `Ctrl+C` to exit `kafka-console-consumer`.

OPTIONAL: Working with record keys

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

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

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

2. Enter a few values such as:

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

3. Press `Ctrl+D` to exit the producer.

4. Now run the **Consumer** with additional arguments to print the key as well as the value:

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

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

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

5. Press `Ctrl+C` to exit the consumer.

The ZooKeeper Shell

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

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

WATCHER::

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

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


```
ls /
[admin, brokers, cluster, config, consumers, controller,
controller_epoch, isr_change_notification, latest_producer_id_block,
log_dir_event_notification, zookeeper]
```

3. Type `ls /brokers` to see this next level of the directory structure.

```
ls /brokers
[ids, seqid, topics]
```

4. Type `ls /brokers/ids` to see the broker ids for the Kafka cluster.

```
ls /brokers/ids
[101]
```

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

5. Type `get /brokers/ids/101` to see the metadata for broker 101.

```
get /brokers/ids/101
{"listener_security_protocol_map":{"PLAINTEXT":"PLAINTEXT"},"endpoint
s":["PLAINTEXT://kafka:9092"],"jmx_port":-
1,"host":"kafka","timestamp":"1581126250804","port":9092,"version":4}
```

6. Type `get /brokers/topics/testing/partitions/0/state` to see the metadata for partition 0 of topic `testing`.

```
get /brokers/topics/testing/partitions/0/state
{"controller_epoch":1,"leader":101,"version":1,"leader_epoch":0,"isr"
:[101]}
```

Note: During client startup, it requests cluster metadata from a broker in the `bootstrap.servers` list. The output of the two previous commands reflects a bit of this cluster metadata included in the broker response. We will cover this metadata request in more detail later in this course.

7. Press `Ctrl+D` to exit the ZooKeeper shell.

Conclusion

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

STOP HERE. THIS IS THE END OF THE EXERCISE.

Lab 02 Producing Messages to Kafka

a. Kafka Producer (Java, C#, Python)

The goal of this lab is to create a simple producer. The producer application reads a file of latitude and longitude values and writes to the topic **`driver-positions`**. See an example file at **`~/confluent-dev/challenge/java-producer/drivers/driver-1.csv`**. For each entry in the file the application writes a Kafka record. When the application reaches the end of the file it loops back to the top. The record is created with the driver id as the key, and the comma separated latitude and longitude as the value. For example:

Key	Value
driver-1	47.5952,-122.3316

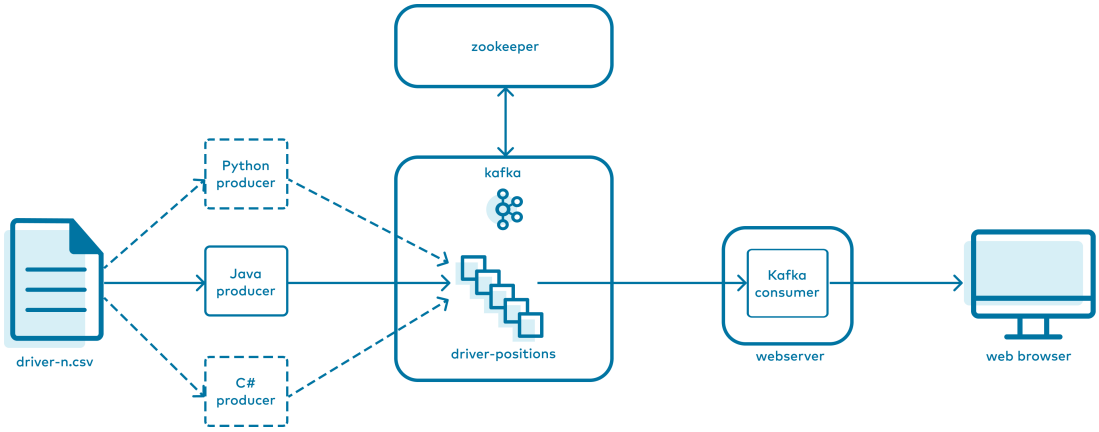
Prerequisites

1. Use the command in the table below to navigate to the project folder for your language. Click the associated hyperlink to open the API reference:

Language	Command	API Reference
Java	**`cd ~/confluent-dev/challenge/java-producer`**	Class KafkaProducer<K,V>
C#	**`cd ~/confluent-dev/challenge/dotnet-producer`**	Interface IProducer<TKey, TValue>
Python	**`cd ~/confluent-dev/challenge/python-producer`**	class confluent_kafka.Producer

2. Run the Kafka cluster:

```
$ docker-compose up -d zookeeper kafka control-center create-topics webserver
```



The `create-topics` container creates the topics for all of the upcoming exercises and then exits. The `webserver` container is running a web application that is consuming from the `driver-positions` and displaying the position of each driver.

3. View the application at http://localhost:3001. You will see a driver appear on the map when you complete the code challenges in this exercise.

4. Run the `kafka-topics` command to see the new topics. All of the new topics are prefixed with `driver`:

```
$ kafka-topics --bootstrap-server kafka:9092 --describe | grep
'driver'
Topic: driver-positions-pyavro  PartitionCount: 3
ReplicationFactor: 1    Configs:
    Topic: driver-positions-pyavro  Partition: 0    Leader: 101
Replicas: 101   Isr: 101    Offline:
    Topic: driver-positions-pyavro  Partition: 1    Leader: 101
Replicas: 101   Isr: 101    Offline:
    Topic: driver-positions-pyavro  Partition: 2    Leader: 101
Replicas: 101   Isr: 101    Offline:
...
```


If any replicas were listed as offline, this would be an indication that the corresponding broker is also offline.

5. If you are completing the C# or Python exercise, install the dependencies.

   a. For C#:

   First, install dotnet running this command. You'll be prompted to enter the password: **training**

```
$ ~/confluent-dev/dotnet-install.sh
```

Then, run:

```
$ dotnet restore
```

b. For Python:

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

6. Open the project in Visual Studio Code. As always, make sure you open VS Code in the correct project folder as specified in step 1.

```
$ code .
```


Writing the Producer

1. Open the implementation file for your language of choice
   - Java `src/main/java/clients/Producer.java`
   - C#: `Program.cs`
   - Python: `main.py`.
2. Locate the `TODO` comments in your implementation file. Use the API reference for your language to attempt each challenge. Solutions are provided at the end of this lab and in the `~/confluent-dev/solution` folder.
3. At any time run the application by selecting the menu **Run** → **Start Debugging** in VS Code. As you complete the challenges try to produce a similar output from your application:

```
Starting Java producer.
Sent Key:driver-1 Value:47.618579,-122.355081
Sent Key:driver-1 Value:47.618577152452055,-122.35520620652974
Sent Key:driver-1 Value:47.61857902704408,-122.35507321130525
Sent Key:driver-1 Value:47.618579488930855,-122.35494018791431
Sent Key:driver-1 Value:47.61857995081763,-122.35480716452278
...
```

Are you having issues with VS code debugging?

Here are some common issues you can check for:

- Did you launch `code` from the correct directory? Ensure you've followed the `cd` command above to change to the project folder.
- Do you miss an earlier step? Quit VS Code, run the missing step, and relaunch VS Code.
- Java users: if VS Code is still having an issue after relaunching try cleaning your workspace. Click the cog wheel icon at the bottom left of VS Code, click **Command Palette**, search for and select **Java: Clean the Java language server workspace**, click **Restart and delete**.

4. When you have the application producing data, leave the application running, and return to the web application at http://localhost:3001.

If you are interested in the inner workings of the web application the source code is available at `~/confluent-dev/webserver`. The web application is a simple Node.js Express web application that contains a Kafka consumer reading from a topic and delivering the records via a Socket.IO websocket to the front end.

5. Use the `kafka-console-consumer` tool to view the data on the `driver-positions` topic:

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

Exit the consumer with `Ctrl+C`.

6. When you have completed the challenges, stop the debugger in VS Code.

Extra Challenges and Questions

1. The producer application takes the driver ID from an environment variable **`DRIVER_ID`**. From new terminal windows run the producer with several different driver IDs. Observe the web application at http://localhost:3001. When you have finished exit your producers with **`Ctrl+C`**.

   a. Java

   ```
   $ cd ~/confluent-dev/solution/java-producer && \
      DRIVER_ID=driver-3 ./gradlew run --console plain
   ```

   b. C#

   ```
   $ cd ~/confluent-dev/solution/dotnet-producer && \
      DRIVER_ID=driver-3 dotnet run
   ```

   c. Python

   ```
   $ cd ~/confluent-dev/solution/python-producer && \
      DRIVER_ID=driver-3 python3 main.py
   ```

2. Experiment with producer settings of **`batch.size`** (default: 16384) and **`linger.ms`** (default: 0 ms). How would you expect the producer to behave with the settings below? Observe the web application at http://localhost:3001.

   a. Java

   ```
   settings.put(ProducerConfig.BATCH_SIZE_CONFIG, 16384);
   settings.put(ProducerConfig.LINGER_MS_CONFIG, 5000);
   ```

   b. C#

   ```
   BatchNumMessages = 16384,
   LingerMs = 5000
   ```

   c. Python


```
'batch.num.messages': 16384,
'linger.ms': 5000
```


Java Solution

solution/java-producer/src/main/java/clients/Producer.java

```
// TODO: configure the location of the bootstrap server
settings.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "kafka:9092");
```

```
// TODO: populate the message object
final ProducerRecord<String, String> record = new ProducerRecord
<>(KAFKA_TOPIC, key, value);
```

```
// TODO: write the lat/long position to a Kafka topic
// TODO: print the key and value in the callback lambda
producer.send(record, (md, e) -> {
  System.out.printf("Sent Key:%s Value:%s\n", key, value);
});
```

C# Solution

solution/dotnet-producer/Program.cs

```
// TODO: configure the location of the bootstrap server
BootstrapServers = "kafka:9092",
```

```
// TODO: populate the message object
var message = new Message<string, string> { Key = driverId, Value = line
};
```

```
// TODO: write the lat/long position to a Kafka topic
// TODO: configure handler as a callback to print the key and value
producer.Produce(KafkaTopic, message, handler);
```

Python Solution

solution/python-producer/main.py

```
#TODO: configure the location of the bootstrap server
'bootstrap.servers': 'kafka:9092',
```

```
#TODO: write the lat/long position to a Kafka topic
#TODO: configure delivery_report as a callback to print the key and
value
producer.produce(
    KAFKA_TOPIC,
    key=DRIVER_ID,
    value=line,
    on_delivery=delivery_report)
```

Extra Challenges and Questions Solutions

1. If you followed the steps successfully you will see multiple cars driving on the map.
2. With the supplied settings for **`batch.size`** and **`linger.ms`** the consumer is batching up records. The amount of batched data never exceeds the **`batch.size`**, so the batch is sent to the broker when the **`linger.ms`** of seconds is met. From the producer documentation:

> linger.ms: ...This setting gives the upper bound on the delay for batching: once we get batch.size worth of records for a partition it will be sent immediately regardless of this setting, however if we have fewer than this many bytes accumulated for this partition we will 'linger' for the specified time waiting for more records to show up...

STOP HERE. THIS IS THE END OF THE EXERCISE.

Lab 04 Consuming Messages from Kafka

a. Kafka Consumer (Java, C#, Python)

The goal of this lab is to create a simple Kafka consumer, that consumes records from the `driver-positions` topic.

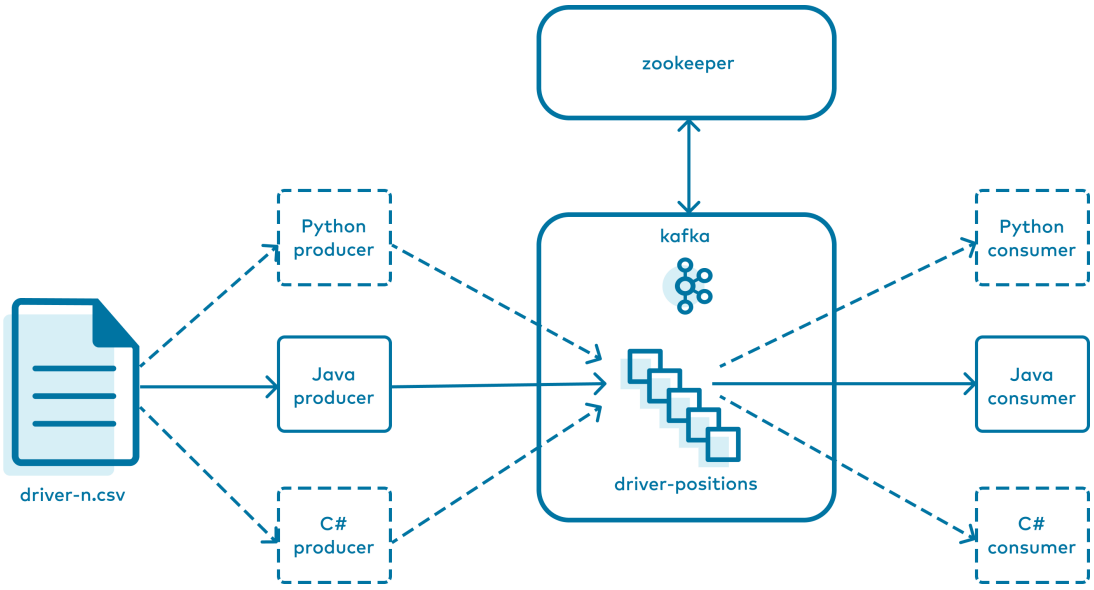
Prerequisites

1. Use the command in the table below to navigate to the project folder for your language. Click the associated hyperlink to open the API reference:

Language	Command	API Reference
Java	`cd ~/confluent-dev/challenge/java-consumer`	Class KafkaConsumer<K,V>
C#	`cd ~/confluent-dev/challenge/dotnet-consumer`	Interface IConsumer<TKey, TValue>
Python	`cd ~/confluent-dev/challenge/python-consumer`	class confluent_kafka.Consumer

2. Run the Kafka cluster:

```
$ docker-compose up -d zookeeper kafka control-center create-topics webserver
```



3. If you are completing the C# or Python exercise, install the dependencies.

   a. For C#:

   ```
   $ dotnet restore
   ```

   b. For Python:

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

4. Open the project in Visual Studio Code:

```
$ code .
```


Writing the Consumer

1. Run the producer solution from the previous exercise in a terminal window. This will give you live data in the `driver-positions` topic. From a terminal window run:

```
$ cd ~/confluent-dev/solution/java-producer && \
  ./gradlew run --console plain
```

2. Open the implementation file for your language of choice

   - Java: `src/main/java/clients/Consumer.java`

- C#: `Program.cs`
- Python: `main.py`.

3. Locate the `TODO` comments in your implementation file. Use the API reference for your language to attempt each challenge. Solutions are provided at the end of this lab and in the `~/confluent-dev/solution` folder.

4. At any time run the application by selecting the menu **Run** → **Start Debugging** in VS Code. As you complete the challenges try to produce a similar output from your application:

```
Starting Java Consumer.
Key:driver-1 Value:47.618579,-122.355081 [partition 1]
Key:driver-1 Value:47.618577152452055,-122.35520620652974 [partition
1]
Key:driver-1 Value:47.61857902704408,-122.35507321130525 [partition
1]
Key:driver-1 Value:47.618579488930855,-122.35494018791431 [partition
1]
Key:driver-1 Value:47.61857995081763,-122.35480716452278 [partition
1]
...
```

5. Leave your consumer running. In a terminal window run this command to inspect the status of your consumer group:

```
$ kafka-consumer-groups \
    --bootstrap-server kafka:9092 \
    --describe \
    --group java-consumer \
    --group csharp-consumer \
    --group python-consumer

GROUP           TOPIC            PARTITION  CURRENT-OFFSET  LOG-END-
OFFSET  LAG ...
csharp-consumer driver-positions 0          -               0
-
csharp-consumer driver-positions 1          2148            2153
5
csharp-consumer driver-positions 2          -               0
-
```

You can see some interesting metrics for your topic consumption. `CURRENT-OFFSET` is the last *committed* offset from your consumers. `LOG-END-OFFSET` is the last offset in each partition. `LAG` is *how far behind* the consumption is, or in other words `LOG-END-OFFSET -`

`CURRENT-OFFSET`. These metrics are very useful when checking if your consumption is keeping up with production.

6. When you have completed the challenges, stop the debugger in VS Code.
7. Return to the terminal window running the producer solution. Press `Ctrl+C` to exit the producer.

Extra Challenges and Questions

1. End your processing, and launch the consumer again. You'll see that the second time you run the application processing begins from a non-zero offset. Does `auto.offset.reset` apply the second time the application is run?
2. How do consumers know where to begin their processing?
3. Can you think of a way to make your next run of the application begin at the offset at the start of each partition?
4. Experiment with consumer settings of `fetch.max.wait.ms` (default: 500ms) and `fetch.min.bytes` (default: 1 byte). How would you expect the consumer to behave with the settings below?

   a. Java

   ```
   settings.put(ConsumerConfig.FETCH_MAX_WAIT_MS_CONFIG, "5000");
   settings.put(ConsumerConfig.FETCH_MIN_BYTES_CONFIG, "5000000");
   ```

   b. C#

   ```
   FetchWaitMaxMs = 5000,
   FetchMinBytes = 5000000,
   ```

   c. Python

   ```
   "fetch.wait.max.ms": "5000",
   "fetch.min.bytes": "5000000"
   ```


Java Solution

solution/java-consumer/src/main/java/clients/Consumer.java

```
// TODO: Poll for available records
final ConsumerRecords<String, String> records = consumer.poll(Duration
.ofMillis(100));
```

```
// TODO: print the contents of the record
System.out.printf("Key:%s Value:%s [partition %s]\n",
    record.key(), record.value(), record.partition());
```

C# Solution

solution/dotnet-consumer/Program.cs

```
// TODO: Consume available records
var cr = consumer.Consume(cts.Token);
```

```
// TODO: print the contents of the record
Console.WriteLine($"Key:{cr.Message.Key} Value:{cr.Message.Value}
[partition {cr.Partition.Value}]");
```


Python Solution

solution/python-consumer/main.py

```
#TODO: Poll for available records
msg = consumer.poll(1.0)
```

```
#TODO: print the contents of the record
print("Key:{} Value:{} [partition {}]".format(
    msg.key().decode('utf-8'),
    msg.value().decode('utf-8'),
    msg.partition()
))
```

Extra Challenges and Questions Solutions

1. `auto.offset.reset` would not be used on the second launch of your consumer. `auto.offset.reset` is used when there is no committed position (e.g. the group is first

initialized) or when an offset is out of range.

2. An instance in a consumer group sends its offset commits and fetches to a group coordinator broker. The group coordinators read from and write to special compacted Kafka topic named *__consumer_offsets*.

   Curious about the *__consumer_offsets* topic? You can consume message on this topic while your consumer runs with the command below:

```
$ kafka-console-consumer --bootstrap-server kafka:9092 \
--topic __consumer_offsets \
--formatter
"kafka.coordinator.group.GroupMetadataManager\$OffsetsMessageFormatte
r" \
| grep 'driver-positions'
```

3. You could begin consumption at the start of each partition simply by updating your consumer to use a new `GROUP_ID`. Also, the utility `kafka-consumer-groups` has a parameter `--to-earliest` which will set offsets to earliest offset. In the next exercise we will see how to programmatically *seek* to an offset for consumption.

4. With the supplied settings for `fetch.max.wait.ms` and `fetch.min.bytes` we see results roughly every 5 seconds. From the consumer documentation:

   > fetch.max.wait.ms: The maximum amount of time the server will block before answering the fetch request if there isn't sufficient data to immediately satisfy the requirement given by fetch.min.bytes.

STOP HERE. THIS IS THE END OF THE EXERCISE.

Lab 07 Schema Management in Apache Kafka

a. Schema Registry, Avro Producer and Consumer (Java, C#, Python)

The goal of this lab is to update our simple producer to write to an AVRO serialized topic `driver-positions-avro`. The code is very similar to your previous producer lab. The code will now communicate with Schema Registry to store and retrieve schemas, and will serialize the structured data in AVRO format.

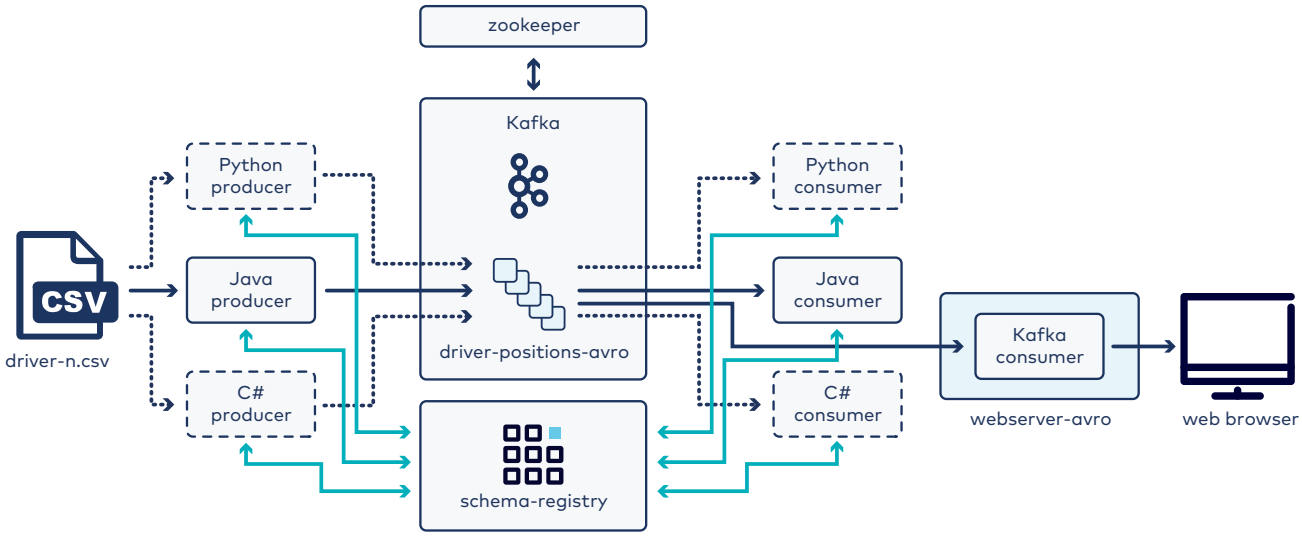
Prerequisites

1. Use the command in the table below to navigate to the project folder for your language:

Language	Command
Java	`cd ~/confluent-dev/challenge/java-producer-avro`
C#	`cd ~/confluent-dev/challenge/dotnet-producer-avro`
Python	`cd ~/confluent-dev/challenge/python-producer-avro`

2. Run the Kafka cluster:

```
$ docker-compose up -d zookeeper kafka control-center create-topics \
    schema-registry webserver-avro
```

You have some new containers you haven't seen before, Schema Registry and an updated version of the web application that can deserialize AVRO serialized data.

Writing the Avro Producer

1. If you are completing the **Java** exercise:
   a. Inspect the schema file at `src/main/avro/position_value.avsc`.
   b. The supplied `build.gradle` file contains Avro plugin `com.commercehub.gradle.plugin.avro` which includes a task `generateAvroJava` to generate POJOs (Plain Old Java Objects/classes) from any Avro schemas in the project.
   c. Use `gradle` to generate the AVRO class:

```
$ ./gradlew build
```

2. If you are completing the **C#** exercise:
   a. Inspect the schema file at `position_value.avsc`.
   b. Install the `avrogen` tool:

```
$ dotnet tool install -g Confluent.Apache.Avro.AvroGen
```

You may need to restart the VM to use `avrogen` in the next step. After restarting, you'll need to run again:

```
$ docker-compose up -d zookeeper kafka control-center
create-topics schema-registry webserver-avro
```

c. Use the `avrogen` to generate the AVRO class:

```
$ avrogen -s position_value.avsc .
```

d. Restore dependencies for your project:

```
$ dotnet restore
```

3. If you are completing the **Python** exercise:

   a. Inspect the schema file at `position_value.avsc`.

   b. Install the dependencies:

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

4. Open the project in Visual Studio Code:

```
$ code .
```

5. Open the implementation file for your language of choice. Can you determine what has changed from the previous producer exercise?

   - Java `src/main/java/clients/Producer.java`
   - C#: `Program.cs`
   - Python: `main.py`

6. Run the application by selecting the menu **Run** → **Start Debugging** in VS Code. You will see your application output:

```
Starting Java Avro producer.
...
Sent Key:driver-1 Latitude:47.618579 Longitude:-122.355081
Sent Key:driver-1 Latitude:47.618577152452055 Longitude:-
122.35520620652974
Sent Key:driver-1 Latitude:47.61857902704408 Longitude:-
122.35507321130525
Sent Key:driver-1 Latitude:47.618579488930855 Longitude:-
122.35494018791431
...
```

7. Leave your Avro producer application running. You can view the web application at http://localhost:3002.
8. Stop the debugger in VS Code.

OPTIONAL: Inspecting the Schema Registry REST API

Next you will inspect the contents and settings of Schema Registry via the REST API. See more details about the API at Schema Registry API Reference.

1. Find all the **subjects** in your Schema Registry:

```
$ curl schema-registry:8081/subjects
["driver-positions-avro-value"]
```

2. How many versions do you see for your subject?

```
$ curl schema-registry:8081/subjects/driver-positions-avro-
value/versions
[1]
```

3. View the contents of version 1 of the schema:

```
$ curl -s schema-registry:8081/subjects/driver-positions-avro-
value/versions/1
{"subject":"driver-positions-avro-
value","version":1,"id":1,"schema":"{\"type\":\"record\",\"name\":\"P
ositionValue\",\"namespace\":\"clients.avro\",\"fields\":[{\"name\":\
"latitude\",\"type\":\"double\"},{\"name\":\"longitude\",\"type\":\"d
ouble\"}]}"}
```

You can get the schema for a specific version of a subject with the `/subjects/(string:`

`subject)/versions/(versionId: version)/schema` path. You can pipe this to `jq` for pretty printing:

```
$ curl -s schema-registry:8081/subjects/driver-positions-avro-
value/versions/1/schema \
  | jq .
{
  "type": "record",
  "name": "PositionValue",
  "namespace": "clients.avro",
  "fields": [
    {
      "name": "latitude",
      "type": "double"
    },
    {
      "name": "longitude",
      "type": "double"
    }
  ]
}
```

4. Our Avro producer self-registered the `driver-positions-avro-value` schema subject when it produced its first record. In a production environment, we would typically have pre-registered the schema subject using the Schema Registry REST API. Let's run the command to do so now and observe the result.

```
$ curl -XPOST  -H "Content-Type:
application/vnd.schemaregistry.v1+json"  schema-
registry:8081/subjects/driver-positions-avro-value/versions/  -d '{
"schema":"{\"type\":\"record\",\"name\":\"PositionValue\",\"namespace
\":\"clients.avro\",\"fields\":[{\"name\":\"latitude\",\"type\":\"dou
ble\"},{\"name\":\"longitude\",\"type\":\"double\"}]}"}'
{"id":1}
```

   The command responds with the schema ID.

5. Check the default compatibility setting:

```
$ curl schema-registry:8081/config
{"compatibilityLevel":"BACKWARD"}
```

Writing the Avro Consumer

1. Use the command in the table below to navigate to the project folder for your language:

Language	Command
Java	`cd ~/confluent-dev/challenge/java-consumer-avro`
C#	`cd ~/confluent-dev/challenge/dotnet-consumer-avro`
Python	`cd ~/confluent-dev/challenge/python-consumer-avro`

2. Complete the same initialization steps you did for the producer exercise.

   a. Java

   ```
   $ ./gradlew build
   ```

   b. C#

   ```
   $ avrogen -s position_value.avsc . ; dotnet restore
   ```

   c. Python

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

3. Open the project in Visual Studio Code:

   ```
   $ code .
   ```

4. Run the application by selecting the menu **Run** → **Start Debugging** in VS Code. You will see your application output:


```
Starting Java Avro Consumer.
Key:driver-1 Latitude:47.618579 Longitude:-122.355081 [partition 1]
Key:driver-1 Latitude:47.618577152452055 Longitude:-
122.35520620652974 [partition 1]
Key:driver-1 Latitude:47.61857902704408 Longitude:-122.35507321130525
[partition 1]
Key:driver-1 Latitude:47.618579488930855 Longitude:-
122.35494018791431 [partition 1]
Key:driver-1 Latitude:47.61857995081763 Longitude:-122.35480716452278
[partition 1]
...
```

Extra Challenges and Questions

1. Inspect the logs of your Schema Registry docker container:

   ```
   $ docker-compose logs schema-registry | grep '/schemas/ids/1'
   ```

   How many requests to `GET /schemas/ids/1` do you see? Can you explain the number of requests?

2. Modify the earlier `curl -XPOST` command to register a new schema version that doesn't meet the current Schema Registry compatibility setting.

3. Advanced challenge: Try adding a field with a default value to your AVRO producer, for example:

   ```
   {"name": "latitude", "type": "double"},
   {"name": "longitude", "type": "double"},
   {"name": "altitude", "type": "double", "default": 0.0}
   ```

   Would this be BACKWARD compatible? Would this be FORWARD compatible? See the documentation for Compatibility Types. Try producing data to your existing topic with a dummy value for altitude (fun fact: Seattle's highest point is 512ft). Can the consumer application or web application still consume from this topic?

Extra Challenges and Questions Solutions

1. A consumer loads a schema when it first sees a record for the schema id, and caches the result for subsequent records.
2. Adding a field without a default value would not meet the BACKWARD compatibility requirement, for example:

```
$ curl -XPOST  -H "Content-Type:
application/vnd.schemaregistry.v1+json"  schema-
registry:8081/subjects/driver-positions-avro-value/versions/  -d '{
"schema":"{\"type\":\"record\",\"name\":\"PositionValue\",\"namespace
\":\"clients.avro\",\"fields\":[{\"name\":\"latitude\",\"type\":\"dou
ble\"},{\"name\":\"longitude\",\"type\":\"double\"},{\"name\":\"new_f
ield\",\"type\":\"double\"}]}"}'
{"error_code":409,"message":"Schema being registered is incompatible
with an earlier schema"}
```

3. Adding a field with a default value is both FORWARD and BACKWARD compatible. If you were to produce data to the `driver-positions-avro` topic with a value for altitude consumers built with version 1 of the schema would ignore the values for altitude, making this update FORWARD compatible.

STOP HERE. THIS IS THE END OF THE EXERCISE.

Lab 08 Stream Processing with Kafka Streams

a. Kafka Streams (Java)

The goal of this lab is to perform stateless operations on your `driver-positions-avro` topic to filter out the events from `driver-2`, add a new field in the value containing the Latitude and Longitude in `String` format and write the results to a new topic `driver-positions-string-avro`.

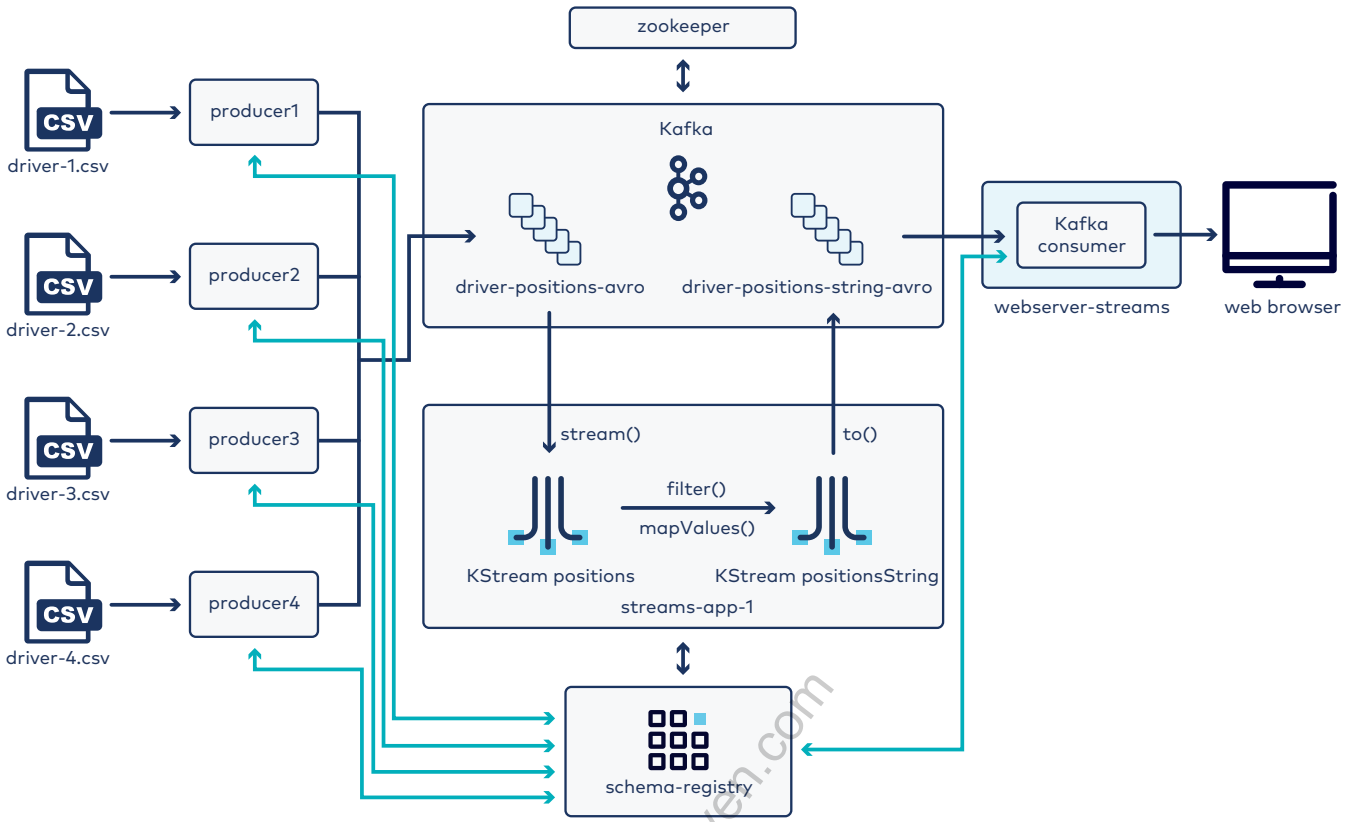
Prerequisites

1. Navigate to the project folder:

```
$ cd ~/confluent-dev/challenge/java-streams-avro
```

2. Run the Kafka cluster:

```
$ docker-compose up -d zookeeper kafka control-center create-topics \
    schema-registry producer1 producer2 producer3 producer4 \
    webserver-avro webserver-streams
```

The new **`producer`** containers are running exactly the same code created in the *Avro Producer* exercise. These containers will simulate four drivers driving around a city. You can see the activity on the **`driver-positions-avro`** topic in the web application http://localhost:3002.

Writing the Streams Processing

1. Open the project in Visual Studio Code:

```
$ code .
```

2. Open the implementation file **`src/main/java/clients/StreamsApp.java`**
3. Let's focus on the topology from Line 66 in the code. See the Kafka Streams DSL documentation for stateless transformations. In this lab, you'll use two transformations, **`filter()`** and **`mapValues()`**.
4. The **`filter()`** method can be defined using a lambda expression where the inputs are the **`key`** and **`value`** of the event. It also requires a predicate/condition to define if the event is filtered or not.

Go to Lines 88-89 in the code and try to define the predicate to filter out events from `driver-2`.

Solution

```
final KStream<String, PositionValue> positionsFiltered =
positions.filter(
        (key,value) -> !key.equals("driver-2"));
```

5. The `mapValues()` method applies a transformation to the value of each event. You can use a lambda expression to define the transformation where the input is just the value. Go to Lines 96-104 and try to complete the missing pieces to change the value from PositionValue type to PositionString type, which contains a new field `positionString`.

Solution

```
final KStream<String, PositionString> positionsString =
positionsFiltered.mapValues(
    value -> {
        final Double latitude = value.getLatitude();
        final Double longitude = value.getLongitude();
        final String positionString = "Latitude: " + String
.valueOf(latitude) +
                                      ", Longitude: " + String
.valueOf(longitude);
        return new PositionString(latitude, longitude,
positionString);
    }
);
```


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

7. View the new topic with the `kafka-avro-console-consumer` tool:

```
$ kafka-avro-console-consumer --bootstrap-server kafka:9092 \
    --property schema.registry.url=http://schema-registry:8081 \
    --topic driver-positions-string-avro --property print.key=true \
    --key
-deserializer=org.apache.kafka.common.serialization.StringDeserialize
r \
    --from-beginning
```

8. Visit the web application subscribed to `driver-positions-string-avro` at

http://localhost:3003/

9. Stop the debugger in VS Code.

STOP HERE. THIS IS THE END OF THE EXERCISE.

Lab 09 Event Streaming Apps with ksqlDB

The goal of this lab is to build an augmented (or enriched) topic from our sources of driver and position data. This position data is created by our AVRO Kafka producer, that we created earlier. The driver data is delivered manually to Kafka using the tool `kafka-console-producer`.

a. ksqlDB - Join a Stream and a Table

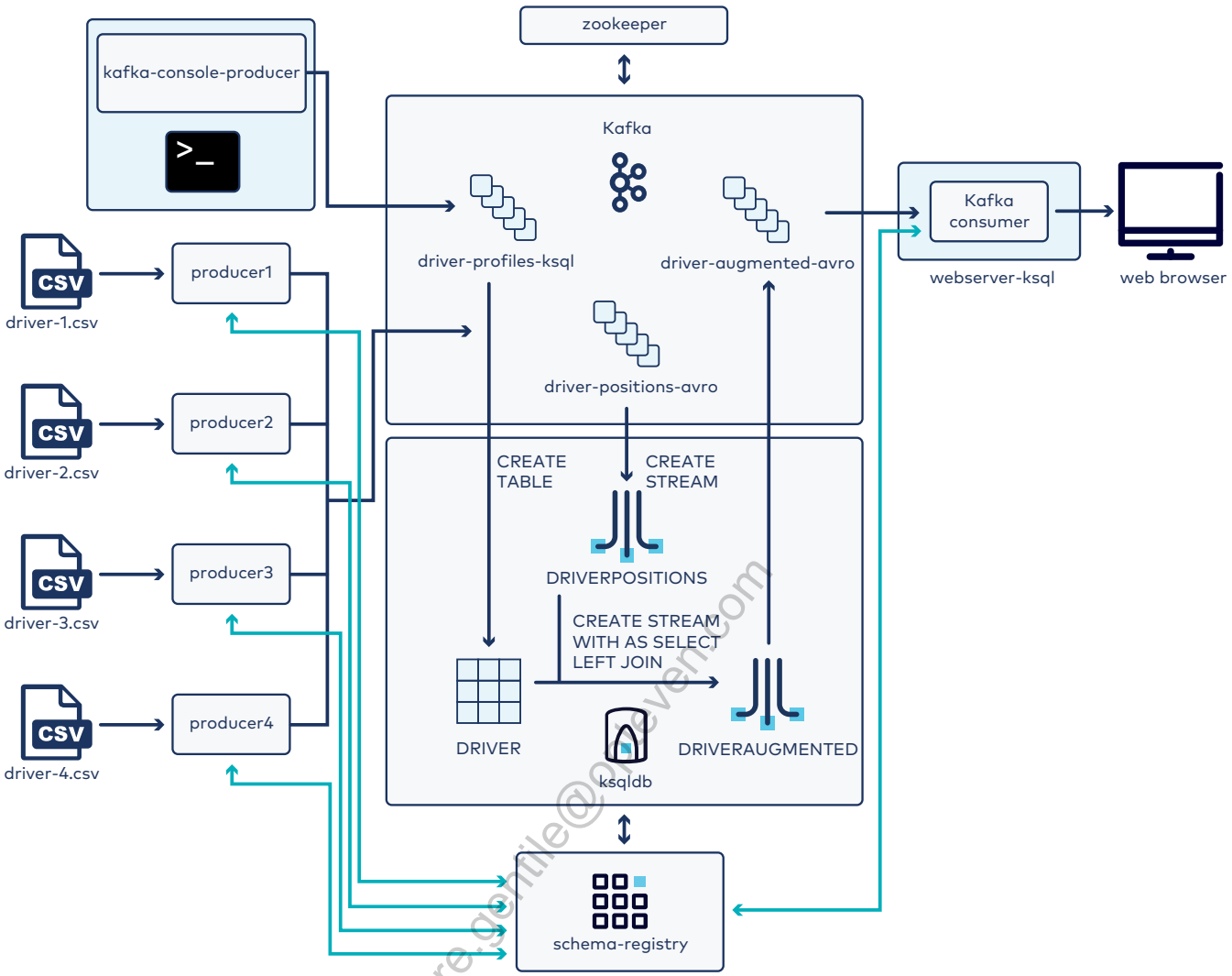
Prerequisites

1. Navigate to the project folder:

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

2. Run the Kafka cluster:

```
$ docker-compose up -d zookeeper kafka control-center \
    schema-registry postgres create-topics \
    producer1 producer2 producer3 producer4 \
    ksqldb-server webserver-ksql
```

You have created a new `ksqldb-server` container, a webserver consuming the new topic, and containers to simulate four drivers driving around a city.

Write Driver Profiles data to Kafka

We use the `kakfa-console-producer` to write information about each driver (name, surname, car model, etc.) to topic `driver-profiles-ksql`.

1. Run this command to start the `kafka-console-producer`:

```
$ kafka-console-producer \
    --bootstrap-server kafka:9092 \
    --topic driver-profiles-ksql \
    --property parse.key=true \
    --property key.separator=:
>
```

2. Now, the producer is waiting for inputs. Copy the messages below and paste them in the Terminal:

```
driver-1:Randall|Palmer|Toyota|Corolla
driver-2:Razı|İnönü|Nissan|Sentra
driver-3:□□□|□□|Subaru|Forester
driver-4:David|Chandler|Tesla|S
```

Note that you are producing messages with Key (driverId). The Value is an entire String with the fields delimited by "|".

3. Exit `kafka-console-producer` pressing `Ctrl+C`.

Create a Table from Drive Profiles data

1. Execute the ksqlDB CLI:

```
$ ksql http://ksqldb-server:8088
...
Copyright 2017-2021 Confluent Inc.

CLI v7.0.0, Server v7.0.0 located at http://ksqldb-server:8088
Server Status: RUNNING

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

ksql>
```

2. Create a table from the topic `driver-profiles-ksqlavro`:

```
ksql> CREATE TABLE DRIVER (driverkey VARCHAR PRIMARY KEY, firstname
VARCHAR, lastname VARCHAR, make VARCHAR, model VARCHAR)
      WITH (KAFKA_TOPIC='driver-profiles-ksql',
VALUE_FORMAT='delimited', VALUE_DELIMITER='|');

 Message
---------------
 Table created
---------------
```

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

```
ksql> SET 'auto.offset.reset' = 'earliest';
Successfully changed local property 'auto.offset.reset' to
'earliest'. Use the UNSET command to revert your change.
```

4. Verify that there is data in the table `DRIVER`:

```
ksql> SELECT * FROM DRIVER EMIT CHANGES;
+-------------+-------------+-------------+----------+-------------+
|DRIVERKEY    |FIRSTNAME    |LASTNAME     |MAKE      |MODEL        |
+-------------+-------------+-------------+----------+-------------+
|driver-2     |Razı         |İnönü        |Nissan    |Narkhede     |
|driver-6     |William      |Peterson     |GM        |Berglund     |
|driver-1     |Randall      |Palmer       |Toyota    |Offset       |
...
```

and stop the query with `Ctrl+C`.

Create a Stream and Table Join

1. Create a stream from the topic `driver-positions-avro`:

```
ksql> CREATE STREAM DRIVERPOSITIONS (driverkey VARCHAR KEY, latitude
DOUBLE, longitude DOUBLE)
      WITH (KAFKA_TOPIC='driver-positions-avro',
VALUE_FORMAT='avro');

 Message
----------------
 Stream created
----------------
```

2. To augment data we can join the `DRIVERPOSITIONS` stream the `DRIVER` tables. Create the join:

```
ksql> CREATE STREAM DRIVERAUGMENTED
      WITH (kafka_topic='driver-augmented-avro', value_format='avro')
      AS
      SELECT
        driverpositions.driverkey AS driverkey,
        driverpositions.latitude,
        driverpositions.longitude,
        driver.firstname,
        driver.lastname,
        driver.make,
        driver.model
      FROM driverpositions
      LEFT JOIN driver on driverpositions.driverkey =
driver.driverkey
      EMIT CHANGES;

 Message
-------------------------------------------
 Created query with ID CSAS_DRIVERAUGMENTED_5
-------------------------------------------
```


This is a **create stream as select** (or CSAS) command that will create a persistent query stream from an existing stream. We are now populating the `driver-augmented-avro` topic with the results of this query. You can confirm this in another terminal window with `kafka-avro-console-consumer`:

```
$ kafka-avro-console-consumer \
    --bootstrap-server kafka:9092 \
    --property schema.registry.url=http://schema-registry:8081 \
    --topic driver-augmented-avro \
    --property print.key=true \
    --key
-deserializer=org.apache.kafka.common.serialization.StringDeserialize
r \
    --from-beginning
```

Press Ctrl+C to exit `kafka-avro-console-consumer`.

3. Return to your ksqlDB CLI and verify the join:

```
ksql> SELECT * FROM DRIVERAUGMENTED EMIT CHANGES;
```

Allow the query to run until it has caught up with the live data coming from the producers. You are now seeing live driver data augmented with *joined* driver profile data. If it is taking a long time for the `SELECT` statement to catch up you can press `Ctrl+C` and enter `SET 'auto.offset.reset' = 'latest';` to set ksqlDB read from the **latest** offset.

Stop the query with `Ctrl+C`. Did you see `null` values in the driver profile columns at the beginning of the results? The query is a `LEFT JOIN` - we will see records from the left side (`DRIVERPOSITIONS`) if there isn't a matching record on the right side (`DRIVER`). Where you see `null` values there isn't a matching record because the position was written to the topic *before* there was a matching entry in `DRIVER` table.

4. Visit the web application consuming from the `driver-augmented-avro` topic at http://localhost:3004/
5. Exit the ksqlDB CLI by pressing `Ctrl+D`.

Extra Challenges and Questions

1. In this exercise the topics `driver-profiles-ksql` and `driver-positions-avro` are joined - what properties do the topics need in common for a successful join?

Extra Challenges and Questions Solutions

1. When you use ksqlDB to join streaming data, you must ensure that your streams and tables are co-partitioned. To be considered co-partitioned we look at 3 properties:
   a. Records have the same keying scheme - both topics are keyed on `driver-id`.
   b. Records have the same number of partitions - both topics have 3 partitions.
   c. Records have the same partitioning strategy - the producers and Kafka Connect are using the default murmur2 hash partitioner.

These properties guarantee that records of the same key will always be delivered to the same partition number in both topics. For example, `driver-3` will be delivered to partition 1 for both `driver-profiles-ksql` and `driver-positions-avro`.

Clean-up

1. Run this command to stop all containers:

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

STOP HERE. THIS IS THE END OF THE EXERCISE.

Lab 11 Data Pipelines with Kafka Connect

a. Kafka Connect - Database to Kafka

The goal of this lab is to build a data pipeline that captures all the changes to a database table `driver-profiles` and writes the changes to a Kafka topic `driver-profiles-avro`. This can all be automated using the Kafka Connect and the JDBC source connector.

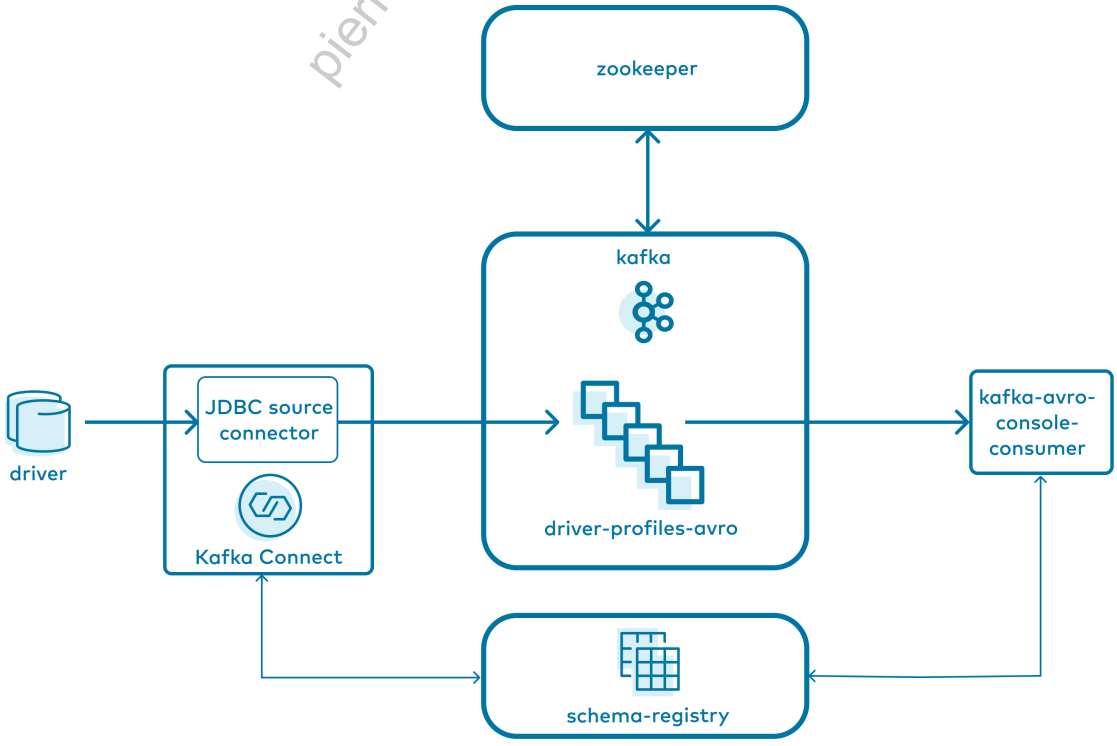
Prerequisites

1. Navigate to the project folder:

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

2. Run the Kafka cluster:

```
$ docker-compose up -d zookeeper kafka control-center create-topics \
    schema-registry connect postgres
```

You have now turned on containers for Kafka Connect and a Postgres database. If you look in `postgres/docker-entrypoint-initdb.d/001-driver.sql` you can see the SQL script used to create and populate the `driver-profiles` table in the Postgres database.

Inspecting Postgres

1. Inspect the contents of the `driver-profiles` table by first connecting to the Postgres database:

```
$ psql -h postgres -U postgres
psql (11.2)
Type "help" for help.

postgres=#
```

2. At the postgres prompt use a SQL select statement to view the contents of the `driver-profiles` table:

```
postgres=# select * from driver;
 id | driverkey | firstname | lastname |  make   |  model   |
timestamp
----+-----------+-----------+----------+---------+----------+
---------------------------
  1 | driver-1  | Randall   | Palmer   | Toyota  | Offset   | 2020-
01-26 01:11:31.707991
  2 | driver-2  | Razı      | İnönü    | Nissan  | Narkhede | 2020-
01-26 01:11:31.709005
...
```

3. Press `Q` to exit the Table View.

4. Exit `psql` by pressing `Ctrl+D`.

Install the Kafka Connect JDBC Connector

We use the Kafka Connect JDBC connector in this exercise so we need to install it on the worker.

1. Install the connector with the following command (and expected response):

```
$ docker-compose exec -u root connect confluent-hub install
confluentinc/kafka-connect-jdbc:10.0.0
The component can be installed in any of the following Confluent
Platform installations:
  1. / (installed rpm/deb package)
  2. / (where this tool is installed)
Choose one of these to continue the installation (1-2):
```

2. At the prompt, type `1` and press **Enter**:

```
Choose one of these to continue the installation (1-2): 1
```

3. You'll be prompted again. At the prompt, type `y` and press **Enter**.

```
Do you want to install this into /usr/share/confluent-hub-components?
(yN) y
```

4. You'll be prompted again. At the prompt, type `y` and press **Enter**.

```
Component's license:
Confluent Community License
https://www.confluent.io/confluent-community-license
I agree to the software license agreement (yN) y
```

5. You'll be prompted again. At the prompt, type `y` and press **Enter**.

```
Downloading component Kafka Connect JDBC 10.0.0, provided by
Confluent, Inc. from Confluent Hub and installing into
/usr/share/java/kafka
Detected Worker's configs:
  1. Standard: /etc/kafka/connect-distributed.properties
  2. Standard: /etc/kafka/connect-standalone.properties
  3. Standard: /etc/schema-registry/connect-avro-
distributed.properties
  4. Standard: /etc/schema-registry/connect-avro-
standalone.properties
  5. Used by Connect process with PID : /etc/kafka-connect/kafka-
connect.properties
Do you want to update all detected configs? (yN) y
```

The installation completes.

```
Adding installation directory to plugin path in the following files:
  /etc/kafka/connect-distributed.properties
  /etc/kafka/connect-standalone.properties
  /etc/schema-registry/connect-avro-distributed.properties
  /etc/schema-registry/connect-avro-standalone.properties
  /etc/kafka-connect/kafka-connect.properties

Completed
```

6. To complete the installation, we need to restart the `connect` container:

```
$ docker-compose restart connect
```

7. Verify that the Connect Worker successfully restarted prior to continuing to the next step:

```
$ docker-compose logs connect | grep -i "INFO .* Finished starting
connectors and tasks"
connect              | [2022-04-07 18:16:59,032] INFO [Worker
clientId=connect-1, groupId=connect] Finished starting connectors and
tasks
(org.apache.kafka.connect.runtime.distributed.DistributedHerder)
connect              | [2022-04-07 18:32:14,011] INFO [Worker
clientId=connect-1, groupId=connect] Finished starting connectors and
tasks
(org.apache.kafka.connect.runtime.distributed.DistributedHerder)
```


Repeat this command until the **Finished starting connectors and tasks** message appears twice.

Configure the AVRO source connector

1. Add a JDBC source connector via command line with the `curl` command below. Let's focus on the transformations that are happening in the connector settings. You can read more about transformations in the documentation for Kafka Connect Transformations.

   a. **RegexRouter** By default the records would be written to a topic with the same name as the table. The setting here will update record topic to `driver-profiles-avro`.

   b. **ValueToKey** The connector is configured to use the `driverkey` property as the record key. At this point the key in the record would look like `{driverkey=driver-5}`.

c. **ExtractField$Key** The connector extracts the `driverkey` field from the key and replaces the entire key with the extracted field. A key of `{driverkey=driver-5}` would be replaced with `driver-5`.

```
$ curl -X POST \
  -H "Content-Type: application/json" \
  --data '{
    "name": "Driver-Connector-Avro",
    "config": {
      "connector.class":
"io.confluent.connect.jdbc.JdbcSourceConnector",
      "connection.url":
"jdbc:postgresql://postgres:5432/postgres",
      "connection.user": "postgres",
      "table.whitelist": "driver",
      "topic.prefix": "",
      "mode":"timestamp+incrementing",
      "incrementing.column.name": "id",
      "timestamp.column.name": "timestamp",
      "table.types": "TABLE",
      "numeric.mapping": "best_fit",
      "key.converter":
"org.apache.kafka.connect.storage.StringConverter",
      "value.converter":
"io.confluent.connect.avro.AvroConverter",
      "value.converter.schema.registry.url": "http://schema-
registry:8081",
      "transforms": "suffix,createKey,extractKey",

"transforms.suffix.type":"org.apache.kafka.connect.transforms.Rege
xRouter",
      "transforms.suffix.regex":"(.*)",
      "transforms.suffix.replacement":"$1-profiles-avro",
      "transforms.createKey.type":
"org.apache.kafka.connect.transforms.ValueToKey",
      "transforms.createKey.fields": "driverkey",
      "transforms.extractKey.type":
"org.apache.kafka.connect.transforms.ExtractField$Key",
      "transforms.extractKey.field": "driverkey"
    }
}' http://connect:8083/connectors
```


the answer should be something like this:

```
{"name":"Driver-Connector-
Avro","config":{"connector.class":"io.confluent.connect.jdbc.JdbcS
ourceConnector","connection.url":"jdbc:postgresql://postgres:5432/
postgres","connection.user":"postgres","table.whitelist":"driver",
"topic.prefix":"","mode":"timestamp+incrementing","incrementing.co
lumn.name":"id","timestamp.column.name":"timestamp","table.types":
"TABLE","numeric.mapping":"best_fit","key.converter":"org.apache.k
afka.connect.storage.StringConverter","value.converter":"io.conflu
ent.connect.avro.AvroConverter","value.converter.schema.registry.u
rl":"http://schema-
registry:8081","transforms":"suffix,createKey,extractKey","transfo
rms.suffix.type":"org.apache.kafka.connect.transforms.RegexRouter"
,"transforms.suffix.regex":"(.*)","transforms.suffix.replacement":
"$1-profiles-
avro","transforms.createKey.type":"org.apache.kafka.connect.transf
orms.ValueToKey","transforms.createKey.fields":"driverkey","transf
orms.extractKey.type":"org.apache.kafka.connect.transforms.Extract
Field$Key","transforms.extractKey.field":"driverkey","name":"Drive
r-Connector-Avro"},"tasks":[],"type":"source"}
```

2. Let's see what we get:

```
$ kafka-avro-console-consumer \
    --bootstrap-server kafka:9092 \
    --property schema.registry.url=http://schema-registry:8081 \
    --topic driver-profiles-avro \
    --property print.key=true \
    --key
-deserializer=org.apache.kafka.common.serialization.StringDeserialize
r \
    --from-beginning
```


and we should see something like this:

```
driver-2    {"id":2,"driverkey":"driver-
2","firstname":"Razı","lastname":"İnönü",...
driver-6    {"id":6,"driverkey":"driver-
6","firstname":"William","lastname":"Peterson",...
driver-10   {"id":10,"driverkey":"driver-
10","firstname":"Aaron","lastname":"Gill",...
...
```

It may take several seconds for the records to appear.

3. Exit the consumer with `Ctrl+C`.

Configure the Protobuf source connector

Schema Registry 5.5 added support for Protobuf and JSON Schema along with Avro. You can now add a connector using a `ProtobufConverter` class. The connector will source the same data from the Postgres database, register a Protobuf schema, and write the Protobuf serialized bytes to the Kakfa topic `driver-profiles-protobuf`.

1. The command below is nearly the same as your previous command. We have changed the: name, value.converter, and topic suffix.

```
$ curl -X POST \
  -H "Content-Type: application/json" \
  --data '{
    "name": "Driver-Connector-Protobuf",
    "config": {
      "connector.class":
"io.confluent.connect.jdbc.JdbcSourceConnector",
      "connection.url": "jdbc:postgresql://postgres:5432/postgres",
      "connection.user": "postgres",
      "table.whitelist": "driver",
      "topic.prefix": "",
      "mode":"timestamp+incrementing",
      "incrementing.column.name": "id",
      "timestamp.column.name": "timestamp",
      "table.types": "TABLE",
      "numeric.mapping": "best_fit",
      "key.converter":
"org.apache.kafka.connect.storage.StringConverter",
      "value.converter":
"io.confluent.connect.protobuf.ProtobufConverter",
      "value.converter.schema.registry.url": "http://schema-
registry:8081",
      "transforms": "suffix,createKey,extractKey",

"transforms.suffix.type":"org.apache.kafka.connect.transforms.RegexRo
uter",
      "transforms.suffix.regex":"(.*)",
      "transforms.suffix.replacement":"$1-profiles-protobuf ",
      "transforms.createKey.type":
"org.apache.kafka.connect.transforms.ValueToKey",
      "transforms.createKey.fields": "driverkey",
      "transforms.extractKey.type":
"org.apache.kafka.connect.transforms.ExtractField$Key",
      "transforms.extractKey.field": "driverkey"
    }
}' http://connect:8083/connectors
```


the answer should be something like this:

```
{"name":"Driver-Connector-
Protobuf","config":{"connector.class":"io.confluent.connect.jdbc.Jdbc
SourceConnector","connection.url":"jdbc:postgresql://postgres:5432/po
stgres","connection.user":"postgres","table.whitelist":"driver","topi
c.prefix":"","mode":"timestamp+incrementing","incrementing.column.nam
e":"id","timestamp.column.name":"timestamp","table.types":"TABLE","nu
meric.mapping":"best_fit","key.converter":"org.apache.kafka.connect.s
torage.StringConverter","value.converter":"io.confluent.connect.proto
buf.ProtobufConverter","value.converter.schema.registry.url":"http://
schema-
registry:8081","transforms":"suffix,createKey,extractKey","transforms
.suffix.type":"org.apache.kafka.connect.transforms.RegexRouter","tran
sforms.suffix.regex":"(.*)","transforms.suffix.replacement":"$1-
profiles-protobuf
","transforms.createKey.type":"org.apache.kafka.connect.transforms.Va
lueToKey","transforms.createKey.fields":"driverkey","transforms.extra
ctKey.type":"org.apache.kafka.connect.transforms.ExtractField$Key","t
ransforms.extractKey.field":"driverkey","name":"Driver-Connector-
Protobuf"},"tasks":[],"type":"source"}
```

2. The results will look the same as the AVRO topic. This is because **kafka-protobuf-console-consumer** is deserializing the bytes.

```
$ kafka-protobuf-console-consumer \
    --bootstrap-server kafka:9092 \
    --property schema.registry.url=http://schema-registry:8081 \
    --topic driver-profiles-protobuf \
    --property print.key=true \
    --key
-deserializer=org.apache.kafka.common.serialization.StringDeserialize
r \
    --from-beginning
```

3. Exit the consumer with **Ctrl+C**.

4. The bytes in the **driver-profiles-protobuf** topic are Protobuf serialized. We can see the raw bytes using **kafkacat** and piping the results to **hexdump**. See more about kafkacat at https://github.com/edenhill/kafkacat. From the kafkacat documentation:

> kafkacat is a generic non-JVM producer and consumer for Apache Kafka >=0.8, think of it as a netcat for Kafka.

5. Run the command below to see the raw bytes of one message on the **driver-profiles-protobuf** topic. Let's focus on the options we will be using:

`-b` Bootstrap broker(s).

`-C` Consume mode.

`-c1` Exit after consuming 1 message.

`-t` Topic to consume from.

```
$ kafkacat -b kafka:9092 -C -c1 -t driver-profiles-protobuf | hexdump
-C
00000000  00 00 00 00 03 00 08 09  12 08 64 72 69 76 65 72
|..........driver|
00000010  2d 39 1a 06 e5 8a a0 e5  a5 88 22 06 e5 b0 8f e6  |-
9........".....|
00000020  9e 97 2a 02 47 4d 32 08  42 65 72 67 6c 75 6e 64
|..*.GM2.Berglund|
00000030  3a 0c 08 af dd bf f6 05  10 c0 e2 b9 e3 01 0a
|:..............|
0000003f
```

The wire format documentation explains the format of the bytes. The 0th byte is `00` for the format version number. The next 4 bytes `00 00 00 03` tell us the schema id. The remaining bytes are the Protobuf serialized data.

6. For comparison you can inspect the raw bytes on the `driver-profiles-avro` topic. You can see the the format version number, schema id and *AVRO* serialized data.

```
$ kafkacat -b kafka:9092 -C -c1 -t driver-profiles-avro | hexdump -C
00000000  00 00 00 00 01 12 10 64  72 69 76 65 72 2d 39 0c
|.......driver-9.|
00000010  e5 8a a0 e5 a5 88 0c e5  b0 8f e6 9e 97 04 47 4d
|..............GM|
00000020  10 42 65 72 67 6c 75 6e  64 ae 8b a2 a0 ce 5c 0a
|.Berglund.....\.|
00000030
```

7. We can request the contents of schema just created via the Schema Registry REST API:

```
$ curl schema-registry:8081/subjects/driver-profiles-protobuf-
value/versions/1/schema
syntax = "proto3";

import "google/protobuf/timestamp.proto";

message driver {
  int32 id = 1;
  string driverkey = 2;
  string firstname = 3;
  string lastname = 4;
  string make = 5;
  string model = 6;
  google.protobuf.Timestamp timestamp = 7;
}
```


Extra Challenges and Questions

1. Leave the `kafka-avro-console-consumer` reading from the `driver-profiles-avro` topic in a terminal window. Use `psql` to update a row in the `driver` table, and see the update appear on the `driver-profiles-avro` topic. Hint: the `timestamp` column will need to be updated for connect to detect the changes, set timestamp to `now()` for the current date time.
2. Can you use `kafka-topics` to determine the log cleaning policy for the `driver-profiles-avro` topic? Why would this policy have been chosen?

Extra Challenges and Questions Solutions

1. This `UPDATE` statement in `psql` will update a single row in the drivers table:

   ```
   UPDATE driver SET firstname='Bill', timestamp=now() WHERE id = 6;
   ```

   You will see the update appear on the `driver-profiles-avro` topic:

   ```
   driver-6 {"id":6,"driverkey":"driver-
   6","firstname":"Bill","lastname":"Peterson","make":"GM","model":"Berg
   land","timestamp":"1584128781527"}
   ```

2. We can see the topic property with `kafka-topics`:

   ```
   $ kafka-topics --bootstrap-server kafka:9092 --describe --topic
   driver-profiles-avro
   Topic: driver-profiles-avro PartitionCount: 3   ReplicationFactor: 1
   Configs: cleanup.policy=compact
   ```

   Log compaction means we will always retain at least the last known value for each message. In a follow up exercise we will create a ksqlDB table, a table is an abstraction of a changelog stream. We are only interested in the most recent record for a key, and can clean out any previous values.

Conclusion

In this exercise you have used a Kafka Connect JDBC source connector to import data

residing in PostgreSQL database into the topics `driver-profiles-avro` and `driver-profiles-protobuf` in the Kafka cluster. This data can now, e.g., be used to enrich our driver position data in the next exercise.

STOP HERE. THIS IS THE END OF THE EXERCISE.

Lab 12 Challenges with Offsets

a. Kafka Consumer - offsetsForTimes (Java, C#, Python)

The goal of this lab is to update the consumer application to begin the consumption from a date time setting.

Prerequisites

1. Use the command in the table below to navigate to the project folder for your language. Click the associated hyperlink to open the API reference:

Language	Command	API Reference
Java	`cd ~/confluent-dev/challenge/java-consumer-prev`	offsetsForTimes
C#	`cd ~/confluent-dev/challenge/dotnet-consumer-prev`	OffsetsForTimes
Python	`cd ~/confluent-dev/challenge/python-consumer-prev`	offsets_for_times

2. Run the Kafka cluster:

```
$ docker-compose up -d zookeeper kafka control-center create-topics webserver
```

3. If you are completing the C# or Python exercise, install the dependencies.

   a. For C#:

   ```
   $ dotnet restore
   ```

   b. For Python:

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

4. Open the project in Visual Studio Code:

```
$ code .
```

Writing the Consumer with offsetsForTimes

1. First create at least 5 minutes of data in the `driver-positions` topic. Run the producer solution in a terminal window for at least 5 minutes:

```
$ cd ~/confluent-dev/solution/java-producer && \
  ./gradlew run --console plain
```

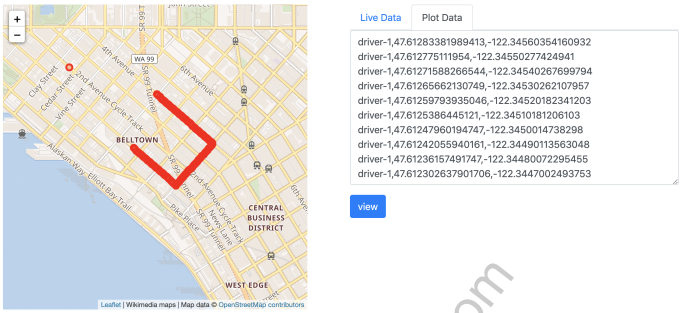
2. Open the implementation file for your language of choice:
   - Java: `src/main/java/clients/Consumer.java`
   - C#: `Program.cs`
   - Python: `main.py`
3. Locate the `TODO` comments in your implementation file. Use the API reference for your language to attempt each challenge. Solutions are provided at the end of this lab and in the `~/confluent-dev/solution` folder.
4. At any time run the application by selecting the menu **Run** → **Start Debugging** in VS Code. As you complete the challenges try to produce a similar output from your application:

```
Starting Java Consumer.
Seeking partition 1 to offset 111
driver-1,47.61283381989413,-122.34560354160932
driver-1,47.612775111954,-122.34550277424941
driver-1,47.61271588266544,-122.34540267699794
driver-1,47.61265662130749,-122.34530262107957
...
```

5. Open the web application at http://localhost:3001.
6. The web application can plot the output from your consumer on the map. Copy into your clipboard the comma separated output from your application, for example:

```
driver-1,47.61283381989413,-122.34560354160932
driver-1,47.612775111954,-122.34550277424941
driver-1,47.61271588266544,-122.34540267699794
driver-1,47.61265662130749,-122.34530262107957
```

7. In the web application select the **Plot Data** tab, paste your comma separated output, and click **view**. The map will plot your data points onto the map, giving you a representation of your driver data beginning at a time 5 minutes ago.



8. When you have completed the challenges, close VS Code.
9. Return to the terminal window running the producer solution. Press `Ctrl+C` to exit the producer.

Extra Challenges and Questions

1. In this exercise we are seeking to an offset based on a timestamp. Can you find the method you would use to seek to the beginning of a partition?

Java Solution

solution/java-consumer-prev/src/main/java/clients/Consumer.java

```
// TODO: Request the offsets for the start timestamp
final Map<TopicPartition, OffsetAndTimestamp> startOffsets =
    consumer.offsetsForTimes(timestampsToSearch);
```

```
// TODO: Print the new offset for each partition
System.out.printf("Seeking partition %d to offset %d\n", entry.getKey
().partition(),
    entry.getValue().offset());
```

C# Solution

solution/dotnet-consumer-prev/Program.cs

```
// TODO: Request the offsets for the start timestamp
var offsets = c.OffsetsForTimes(timestamps, TimeSpan.FromMinutes(1));
```

```
// TODO: Print the new offset for each partition
Console.WriteLine($"Moving partion {offset.Partition.Value} to {offset
.Offset.Value}");
```


Python Solution

solution/python-consumer-prev/main.py

```
#TODO: Request the offsets for the start timestamp
new_offsets = konsumer.offsets_for_times(partitions)
```

```
#TODO: Print the new offset for each partition
print("Setting partition {} to offset {}".format(part.partition, part
.offset))
```

Extra Challenges and Questions Solutions

1. Seeking to the beginning of a partition examples are supplied below.

    a. Java

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

b. C#

```
.SetPartitionsAssignedHandler((c, partitions) =>
{
    var offsets = partitions.Select(tp => new
TopicPartitionOffset(tp, Offset.Beginning));
    return offsets;
})
```

c. Python

```
def my_on_assign(konsumer, partitions):
    for p in partitions:
        p.offset = OFFSET_BEGINNING
    konsumer.assign(partitions)
```


STOP HERE. THIS IS THE END OF THE EXERCISE.

Lab 13 Partitioning Considerations

a. Increasing the Topic Partitions

The goal of this lab is to observe the behavior of a topic before and after the number of topic partitions are increased. In a production environment you might be observing a high lag for a consumer group. Recall from the Kafka documentation:

> Kafka is able to provide both ordering guarantees and load balancing over a pool of consumer processes. This is achieved by assigning the partitions in the topic to the consumers in the consumer group so that each partition is consumed by exactly one consumer in the group. By doing this we ensure that the consumer is the only reader of that partition and consumes the data in order. Since there are many partitions this still balances the load over many consumer instances. **Note however that there cannot be more consumer instances in a consumer group than partitions.**

If you wanted to grow a consumer group beyond the number of topic partitions you would need to increase the number of topic partitions. In this exercise we can observe the impact increasing the number of partitions has on key-partition mapping.

Prerequisites

1. Navigate to the project folder:

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

2. For this exercise we want to start from a clean state. First end all running containers, then run the Kafka cluster:

```
$ docker-compose down -v
...
$ docker-compose up -d zookeeper kafka control-center create-topics \
     schema-registry producer1 producer2 producer3 producer4 \
     webserver-avro
```

Observe Key Partitioning

It will take a few moments for the cluster to start producing data. While it is starting let's look at a command to view the key partition mapping for `driver-positions-avro`.

1. Run `kafkacat` in a terminal window to see the many command line options available. Let's focus on the options we will be using:

   `-C` Consume mode.

   `-e` Exit successfully when last message received.

   `-q` Be quiet - we don't want any extra diagnostic information.

   `-b` Bootstrap broker(s).

   `-t` Topic to consume from.

   `-f` Output formatting string - we are only interested in the message key `%k` and partition `%p`.

2. kafkacat will output a line for every record. To get the partition for each key the output is piped to `sort` and `uniq`:

```
$ kafkacat -Ceq \
    -b kafka \
    -X enable.partition.eof=true \
    -t driver-positions-avro \
    -f 'Key:%k Partition:%p\n' \
    | sort | uniq
Key:driver-1 Partition:1
Key:driver-2 Partition:2
Key:driver-3 Partition:1
Key:driver-4 Partition:1
```

3. You can confirm the key to partition mapping by observing the output on the right hand side of the web application at http://localhost:3002.

4. Increase the number of partitions with `kafka-topics`:

```
$ kafka-topics --bootstrap-server kafka:9092 \
    --alter \
    --topic driver-positions-avro \
    --partitions 10
```

Java producers cache metadata and will start writing to the new partitions when the cache is refreshed. This is determined by the `metadata.max.age.ms` setting which defaults to 300000 milliseconds (5 minutes). For this reason it will take a maximum of 5 minutes before we see data being written to the new partitions.

In the mean time we can take a quick deep dive on the default partitioner.

DefaultPartitioner Deep Dive

You can recreate some of the Kafka DefaultPartitioner logic in the Java 11 REPL (Read–Eval–Print Loop). Using this you can see which partitions will be used for the message keys before and after the change in partition numbers.

1. Upgrade Java SDK. You'll be prompted to enter the password: **training**.

```
$ sudo apt install openjdk-11-jdk-headless
```

2. Start the Java 11 REPL with the following command. The command includes some shared JAR files in the class path:

```
$ kafka_bin=$(dirname $(which kafka-console-consumer))
  share_jars="$kafka_bin/../share/java/kafka/*"
  jshell --class-path "$share_jars"
|  Welcome to JShell -- Version 11.0.14.1
|  For an introduction type: /help intro

jshell>
```

3. Import packages for Kafka utilities and character set information.

```
jshell> import org.apache.kafka.common.utils.Utils; import
java.nio.charset.StandardCharsets; import static
org.apache.kafka.common.utils.Utils.toPositive;
```


4. Define a function that take a `key` and `numPartitions` as parameters. The variable `keyBytes` is set with the UTF-8 bytes of the key. With this we can replicate the logic the Kafka client library uses as the default partitioner: hash the key with murmur2 algorithm and modulo it by the number of partitions

```
jshell> int partition(String key, int numPartitions) {
          byte[] keyBytes = key.getBytes(StandardCharsets.UTF_8);
          return toPositive(Utils.murmur2(keyBytes)) % numPartitions;
        }
|  created method partition(String,int)
```

5. Which partition would the default partitioner use for a record with a key of `driver-1` on a topic using 3 partitions:

```
jshell> System.out.println(partition("driver-1", 3))
1
```

6. Which partition would the default partitioner use for a record with a key of `driver-1` on a topic using 10 partitions:

```
jshell> System.out.println(partition("driver-1", 10))
5
```

7. Exit the Java REPL by pressing Ctrl+D.
8. Confirm the partitions driver records are being delivered to in the web application http://localhost:3002.
9. Consume records from `driver-positions-avro` and observe the keys in each partition.


```
$ kafkacat -Ceq \
    -b kafka \
    -X enable.partition.eof=true \
    -t driver-positions-avro \
    -f 'Key:%k Partition:%p\n' \
    | sort | uniq
Key:driver-1 Partition:1
Key:driver-1 Partition:5
Key:driver-2 Partition:2
Key:driver-2 Partition:9
Key:driver-3 Partition:1
Key:driver-3 Partition:8
Key:driver-4 Partition:1
Key:driver-4 Partition:2
```

Note that records with the key `driver-1` are now being delivered to partition 5. You will also notice `driver-1` has records in both partition 1 (from before the resize) and partition 5 (from after the resize).

10. This is the last exercise!

    a. Clean up your environment with:

    ```
    $ cd ~/confluent-dev
    $ docker-compose down -v
    ```

    b. Verify all services have been shut down by checking `docker-compose ps`

    c. If there are any issues with shutting down, run

    ```
    $ docker-nuke.sh
    ```


STOP HERE. THIS IS THE END OF THE EXERCISE.

Appendix A: Running All Labs with Docker

Running Labs in Docker for Desktop

If you have installed Docker for Desktop on your Mac or Windows 10 Pro machine you are able to complete the course by building and running your applications from the command line.

- Increase the memory available to Docker Desktop to a minimum of 6 GiB. See the advanced settings for Docker Desktop for Mac, and Docker Desktop for Windows.
- Follow the instructions at → Preparing the Labs to `git clone` the source code. The exercise source code will now be on your host machine where you can use any editor to complete the exercises or experiment.
- From the location where you cloned the source code, launch a tools container:

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

All the command line instructions will work from the tools container. This container has been preconfigured with all of the tools you use in the exercises, e.g. `kafka-topics`, `gradle`, `dotnet` and `python`.

```
$ docker-compose exec tools bash
root@tools:/#
```

The source code cloned onto your host machine is present in the tools container as a bind mount. You can see the source code in `~/confluent-dev`.

```
root@tools:/# ls ~/confluent-dev/
README.md  challenge  docker-compose.yml  postgres  solution  update-
hosts.sh  webserver  webserver-avro
```

Anywhere you are instructed to open additional terminal windows you can `exec` additional bash shells on the tools container with `docker-compose exec tools bash` on your host machine.

- The `docker` or `docker-compose` instructions are run on your host machine.

Running the Exercise Applications

From the `tools` container you can use command line alternatives to the VS Code steps used in the instructions. Complete the coding exercises with an editor of your choice on your host machine. When instructed to run code using the VS Code debugger, instead run your code from within the tools container using these terminal commands:

- For Java applications: `./gradlew run`
- For C# applications: `dotnet run`
- For Python applications: `python3 main.py`

Where you are instructed to stop debugging in VS code use `Ctrl+C` to end the running exercise.

Getting Started

Let's apply this to get you started with the **Introduction** exercise.

1. Open a two terminal windows. The first window will be used for `docker` and `docker-compose` commands on your host machine. The second window will be used for the commands run in the tools container.
2. In the first terminal window, clone the source code repository to the folder `confluent-dev`:

```
$ git clone --depth 1 --branch 7.0.0-v1.0.5 \
    https://github.com/confluentinc/training-developer-src.git \
    confluent-dev
```

3. Start the Kafka cluster:

```
$ docker-compose up -d zookeeper kafka control-center
```

4. In the second terminal window launch the tools container, and open a bash shell.

```
$ docker-compose up -d tools
$ docker-compose exec tools bash
```

5. The exercise command line instructions can now be run in the tools container. The first command we have in the **Introduction** exercise uses `zookeeper-shell`:

```
root@tools:/# zookeeper-shell zookeeper:2181 ls /brokers/ids
```

6. If we skip ahead to the **Kafka Producer** coding exercise, this begins with a command to change to the challenge directory. You can do this in the second terminal window where you are accessing the tools container. Using the Java challenge as an example:

```
root@tools:/# cd ~/confluent-dev/challenge/java-producer
```

7. The next command launches additional containers. Run this command in the first terminal window:

```
$ docker-compose up -d zookeeper kafka control-center create-topics
webserver
```

8. Complete the source code challenges in `confluent-dev/challenge/java-producer/src/main/java/clients/Producer.java` on your host machine. Now you can build and run in the second terminal window:

```
root@tools:/# cd ~/confluent-dev/challenge/java-producer
root@tools:/# ./gradlew run
```