WA3285 Introduction to Kafka for PGR

Student Labs

Web Age Solutions

Table of Contents

Lab 1 - Understanding Confluent Cloud Clusters

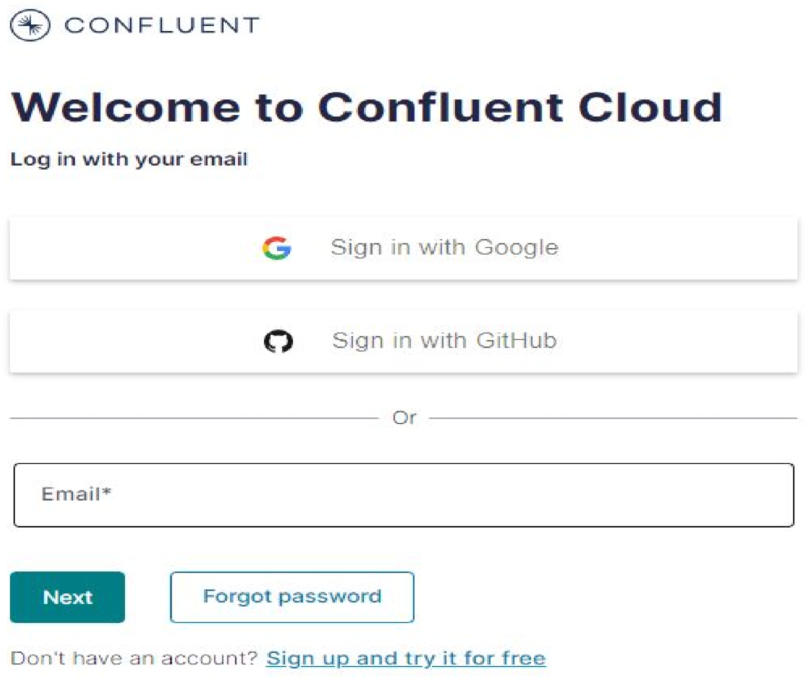
This lab assumes that you have a development account at Confluent Cloud.

Note: The Confluent Cloud Web UI is constantly evolving and despite our best effort to keep up with the changes, you may not see certain UI elements as they were captured at the time of this writing.

Part 1 - Log in to Confluent Cloud

__1. Open your browser and navigate to **https://login.confluent.io/**

__2. Provide your credentials to log in.



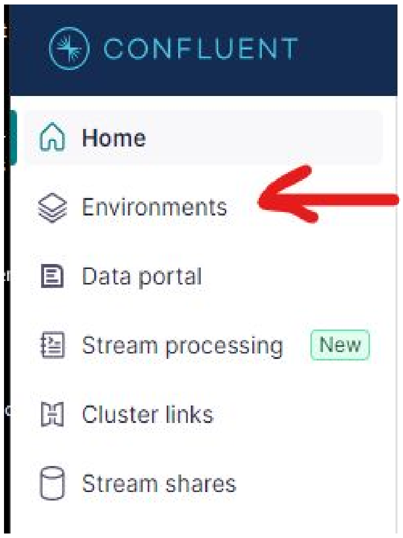
__3. Click **Skip** or otherwise discard any non-essential marketing popups.

Part 2 - Create a Cluster

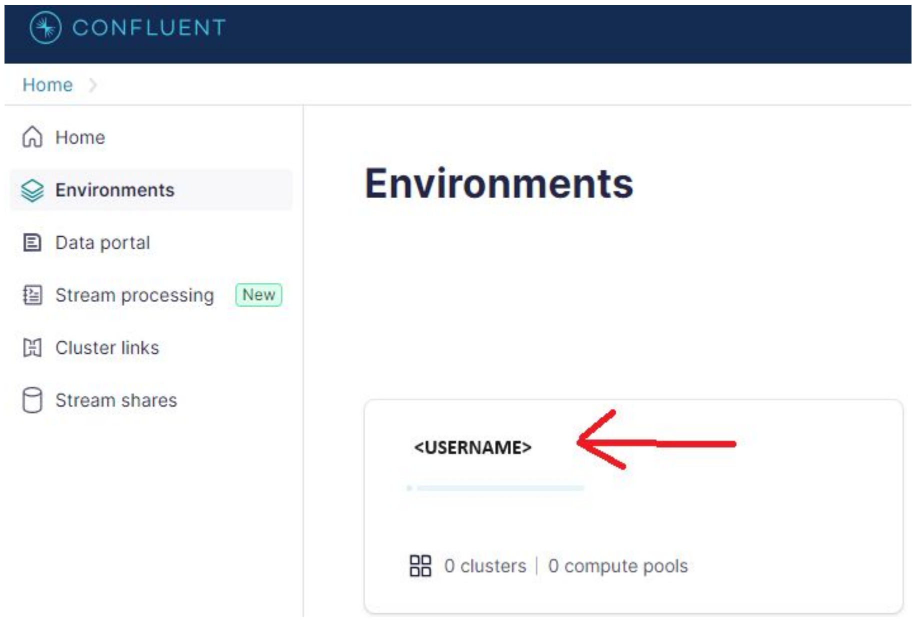
__1. In Confluent Kafka a cluster is a group of servers working together for three reasons: speed (low latency), durability, and scalability. We will add a cluster through the following steps.

__2. From the left hand menu select **Environments.**

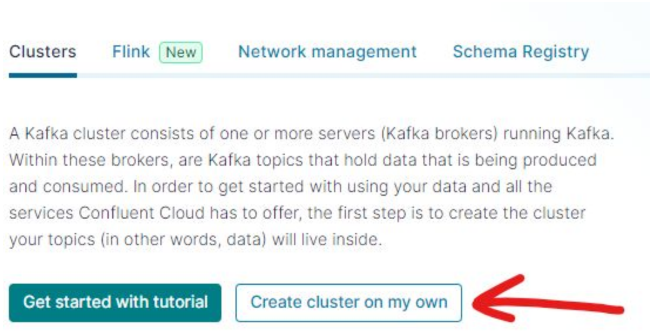
__3.



__4. Click on your "**<username>**" to continue as shown below.

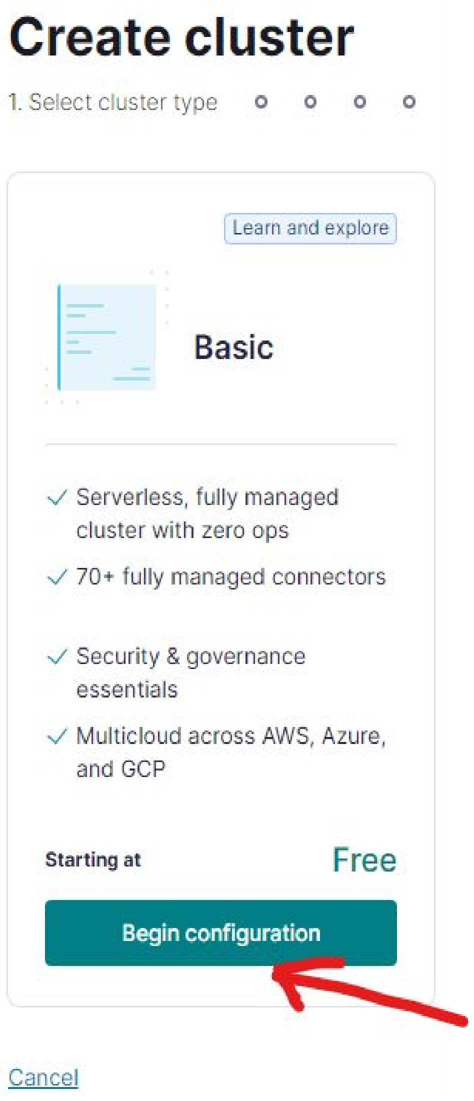


__5. Click **Add Cluster** button *or* if shown, create cluster on my own link.



A Kafka cluster consists of one or more servers (Kafka brokers) running Kafka. Within these brokers, are Kafka topics that hold data that is being produced and consumed. In order to get started with using your data and all the services Confluent Cloud has to offer, the first step is to create the cluster your topics (in other words, data) will live inside.

__6. On the Create cluster page, under the Basic column, click Begin configuration



Create cluster

1. Select cluster type

Learn and explore

Basic

✓ Serverless, fully managed cluster with zero ops

✓ 70+ fully managed connectors

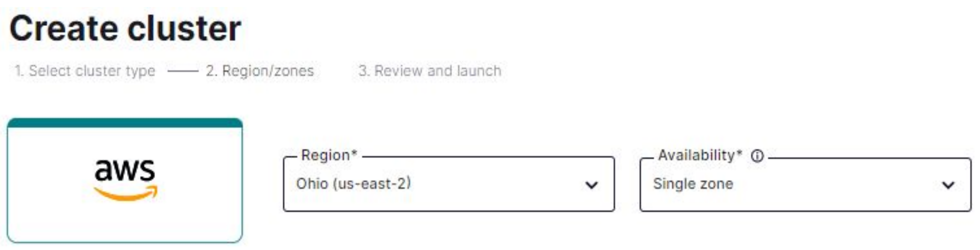
✓ Security & governance essentials

✓ Multicloud across AWS, Azure, and GCP

Starting at Free

__7. Select the Amazon AWS provider and configure the following properties:

- For *Region*, select Ohio **(us-east-2)**
- For *Availability*, select **Single zone**



Create cluster

1. Select cluster type —— 2. Region/zones 3. Review and launch

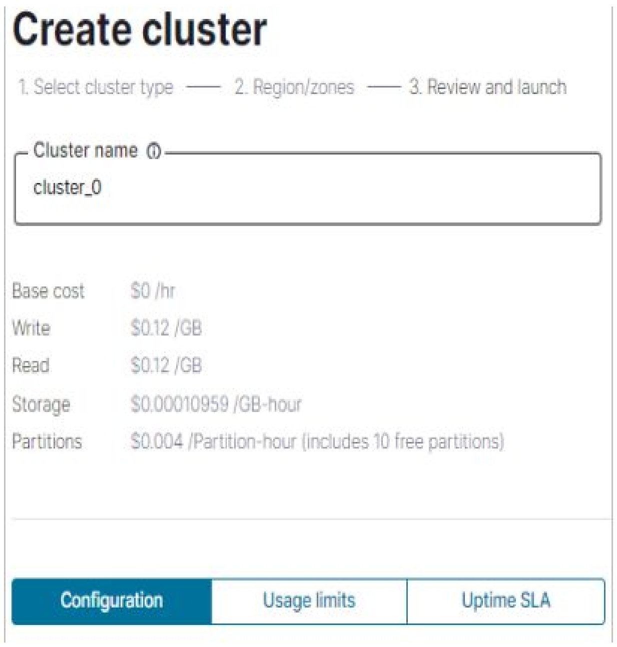
Region*: Ohio (us-east-2)

Availability*: Single zone

__8. Click **Continue.**

__9. You should see the **Create cluster** page

__10.


Create cluster

1. Select cluster type —— 2. Region/zones —— 3. Review and launch

Cluster name: cluster_0

Base cost	$0 /hr
Write	$0.12 /GB
Read	$0.12 /GB
Storage	$0.00010959 /GB-hour
Partitions	$0.004 /Partition-hour (includes 10 free partitions)

__11. Change the name of the **cluster_0** to your own name and initial.

For instance; Bobby Ray would have a cluster named bobby_r

In a production environment it would be advisable to change the default cluster name, cluster_**0**, to something more meaningful in your deployment context. Note that this name is a label (or tag) that helps you with cluster identification.

On this screen you can review the applicable costs, usage limits, and uptime SLA by clicking Review.

The critical piece of information is the **Uptime SLA** metric, which is a guaranteed 99.5% for the **Basic** cluster. In production, you should go with either **Standard** or **Dedicated** cluster options which bumps the SLA to 99.99% -- a more likely level to meet your requirements.

For the **Usage limits** metrics, the following considerations are important:

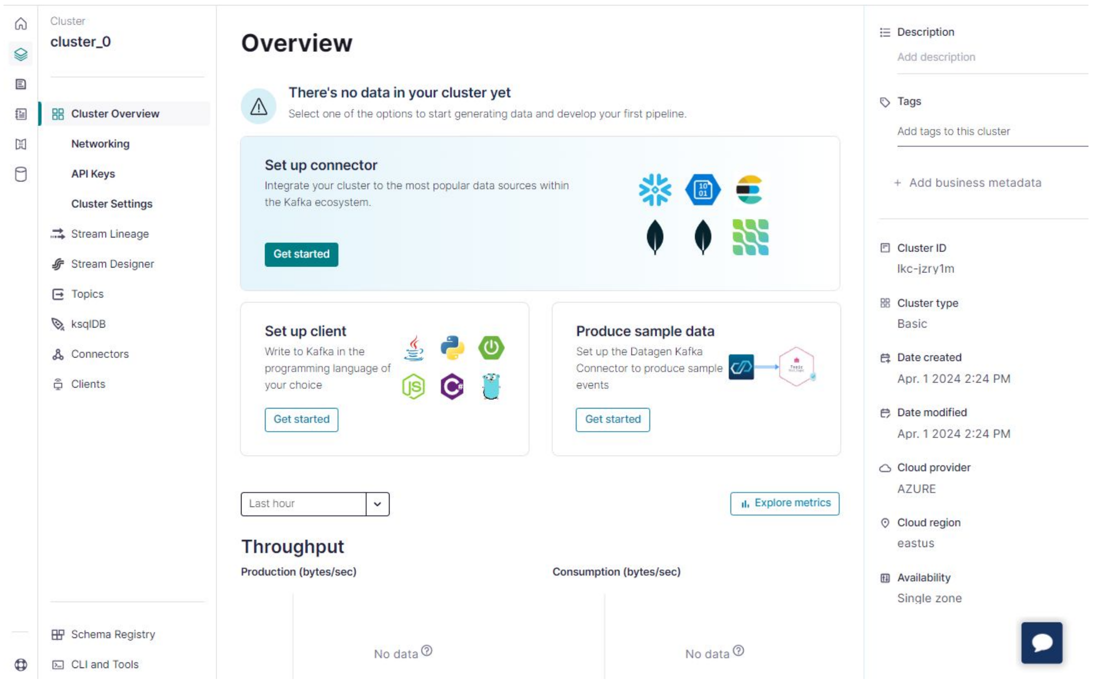
* Max number of partitions, which is mapped to Kafka's processing parallelization capability, is 2K for the **Basic** option (in fact, there are 10 free partitions available for basic and development evaluation use cases).

The **Basic** cluster option gives you support for ~5 MB/s ingress ops and 15 MB/s egress ops per partition with the maximum request size of up to 100MB, meaning that you will need to split larger uploaded objects into smaller chunks to avoid write errors.

Note: Many programming languages offer functionality for sizing in-memory objects; you can use the available object serialization/marshaling mechanism to estimate the object size to be sent over the network.

You may want to review popular platform-neutral structured data serialization schemes such as Protobuf (https://developers.google.com/protocol-buffers) and Avro (https://avro.apache.org/)

__12. Click **Launch cluster**. You should be returned to the main page and see your personally named cluster like cluster_0 herein.



Cluster
cluster_0

Overview

There's no data in your cluster yet
Select one of the options to start generating data and develop your first pipeline.

Set up connector
Integrate your cluster to the most popular data sources within the Kafka ecosystem.

Set up client
Write to Kafka in the programming language of your choice

Produce sample data
Set up the Datagen Kafka Connector to produce sample events

Throughput
Production (bytes/sec)
Consumption (bytes/sec)
No data
No data

Description
Add description

Tags
Add tags to this cluster

Cluster ID
lkc-jzry1m

Cluster type
Basic

Date created
Apr. 1 2024 2:24 PM

Date modified
Apr. 1 2024 2:24 PM

Cloud provider
AZURE

Cloud region
eastus

Availability
Single zone

Your new cluster will be provisioned on the selected cloud platform in the **DEFAULT** Confluent environment.

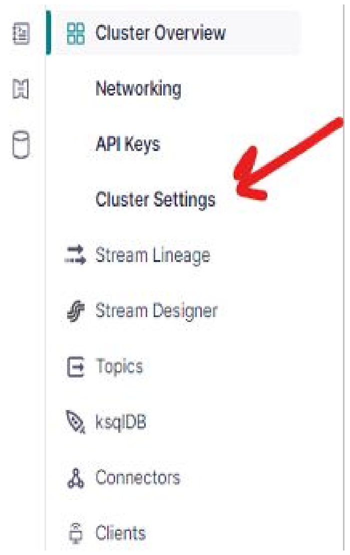
Wait a moment or two while you cluster is prepared and finally up and running.

Generally from this step you could either set up a Kafka connector to start pumping messages in or out of your cluster or set up a Kafka client to get low-level programmatic access to Kafka producer and/or consumer APIs.

Part 3 - Cluster Settings Overview

Let's review the new cluster configuration settings.

__1. From the left hand menu, click **Cluster Settings**

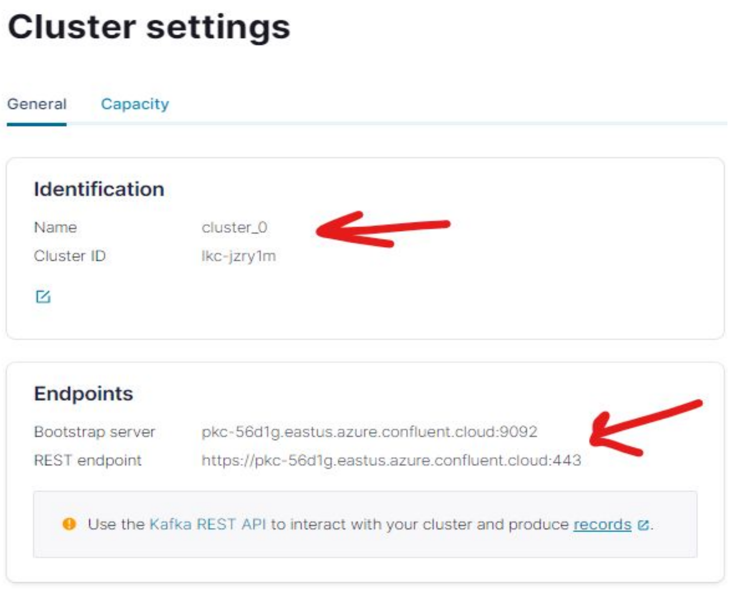


You will be placed on the **Cluster settings** page.

Notice that on this page there is no indication that the cluster is up and running -- once launched, the cluster is assumed to be running until deleted (there is no cluster "stop" command.)

Currently, Confluent Cloud doesn't offer an embedded CLI console and you need to install the Confluent CLI tool (https://docs.confluent.io/confluent-cli/current/overview.html) locally to enable CLI access to your Confluent Cloud deployment.

You should see a summary of your cluster settings similar to below, including the Name and Cluster ID, plus the REST endpoint URI that you can use to interact with your Confluent Cloud using REST-enabled clients.



Identification

Name	cluster_0
Cluster ID	lkc-jzry1m

Endpoints

Bootstrap server	pkc-56d1g.eastus.azure.confluent.cloud:9092
REST endpoint	https://pkc-56d1g.eastus.azure.confluent.cloud:443

Use the Kafka REST API to interact with your cluster and produce records.

__2. You could open a new Notepad and write down the cluster Name, and Cluster Id for later usage. Save it as clusterdetails.txt for instance.

__3. Click **Topics** on the left side menu.

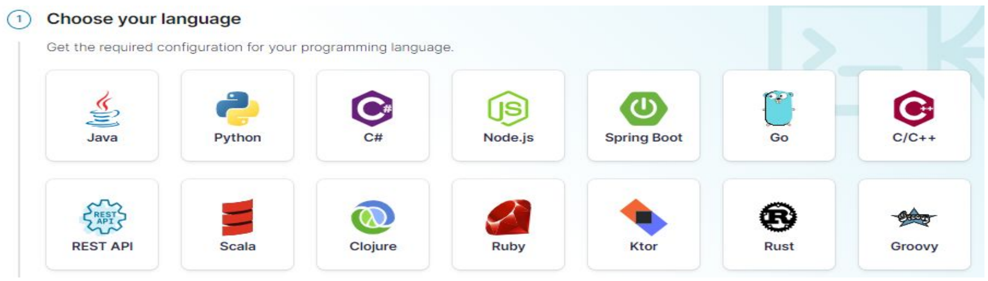
On the **Topics** page, you can create Kafka topics to connect your producer and consumer applications.

We are not going to create Kafka topics in this lab -- it will be a subject of one of the next labs.

__4. Click **Clients** on the left menu.

On the **New Client** page, you should see a list of languages (for the client side) to integrate with Kafka (the server).

Review the available Kafka client options.



__5. Click **Connectors** on the left menu.

You should see a list of the available connectors (libraries) used to integrate Kafka with external data sources (where Kafka acts as the consumer of events/messages produced by the other system) and sinks (where Kafka acts as a producer for events/messages sent to the other system) -- there are 220+ available connectors at the time of this writing.

The Confluent cloud-managed connectors are marked with this icon:

The other category is the "self-managed" connectors.

__6. Click **API keys** on the left menu under Cluster overview.

On this page, you can create Kafka API keys (each of which consists of a key and a secret and is valid for a single Kafka cluster only) that are required to interact with Kafka clusters in Confluent Cloud. Be aware of the cap on how many API keys you can create in your cluster type -- the Basic cluster option sets the limit at 50 API keys per cluster, so try to reuse as many obtained keys as you practically can.

To learn more about API keys, visit https://docs.confluent.io/cloud/current/access-management/authenticate/api-keys/api-keys.html#manage-ccloud-api-keys

This is the last step in this lab.

Part 4 - Review

In this lab, you learned about Kafka clusters in Confluent Cloud.

Lab 2 - Understanding Confluent Cloud CLI

This lab will walk you through the installation process for the Confluent Cloud CLI client (the *confluent* tool); you will also learn how to get started with the Confluent CLI.

The *confluent* tool is a shell that comes with useful features such as command auto-completion; the tool is written in GO.

Part 1 - Confluent CLI Installation

__1. Open your browser and navigate to the Confluent CLI overview page:

```
https://docs.confluent.io/confluent-cli/current/overview.html
```

Familiarize yourself with the key facts about the Confluent CLI tool. Confluent offers setup guides for several operating systems.

__2. Navigate to the link below and review the available installation options.

```
https://docs.confluent.io/confluent-cli/current/install.html#cli-
install
```

In this lab, we will install Confluent CLI v. 3.55.0 (the latest at the time of this writing) on Window® 64-bit. (See this link for documentation on the latest version **`https://github.com/confluentinc/cli/releases/latest )`**

__3. Click on the link for the correct Version to install.

The related bundle will get downloaded to your local machine.

For Confluent CLI v. 3.55.0, the binary name is:

```
confluent_v3.55.0_windows_amd64.zip
```

Note: Version may vary.

__4. Unzip the bundle in a directory of your preference.

The setup is, essentially, complete.

Part 2 - Understanding the Confluent CLI

The confluent tool, *confluent.exe*, is located in:

```
<Your Local Unzipped Dir>\confluent\
```

__1. Open a Command Prompt and change directory to *<Your Local Dir>\confluent\*

__2. Run confluent.exe

You should see the following output:

```
Manage your Confluent Cloud or Confluent Platform. Log in to see all
available commands.

Usage:
  confluent [command]

Available Commands:
  cloud-signup    Sign up for Confluent Cloud.
  completion      Print shell completion code.
  context         Manage CLI configuration contexts.
  help            Help about any command
  kafka           Manage Apache Kafka.
  login           Log in to Confluent Cloud or Confluent Platform.
  logout          Log out of Confluent Cloud or Confluent Platform.
  prompt          Add Confluent CLI context to your terminal prompt.
  shell           Start an interactive shell.
  update          Update the Confluent CLI.
  version         Show version of the Confluent CLI.

Flags:
      --version         Show version of the Confluent CLI.
  -h, --help            Show help for this command.
  -v, --verbose count   Increase verbosity (-v for warn, -vv for info,
 -vvv for debug, -vvvv for trace).

Use "confluent [command] --help" for more information about a command.
```

__3. Enter the following command:

```
confluent login --save
```

__*4. Provide your Confluent Cloud credentials when prompted.*

Note: The *--save* flag will keep you logged in when your login credentials or SSO refresh token expires (which happens after one hour without the *--save s*ub-command).

Once you have successfully authenticated yourself, you can use the CLI.

__5. Enter the following command:

```
confluent environment list
```

You should see the available dev environments (your IDs will differ):

```
       ID       |  Name
```

```
---------------+----------
  * env-j586vp | default
```

Note: Your ID will be different.

The above information should match your data on the **Environment settings** tab for the **default** environment.

__6. Activate your environment with this command (provide your ID below)

```
confluent environment use env-j586vp
```

The tool will print this message:

```
Now using "env-j586vp" as the default (active) environment.
```

Now you can see the clusters added to the environment.

__7. Enter the following command:

```
confluent kafka cluster list
```

You should see the following output:

```
    Id      |    Name    | Type  | Provider |  Region  | Availability | Status
------------+------------+-------+----------+----------+--------------+-------
lkc-do0z5o | cluster_0 | BASIC | azure    | eastus   | single-zone  | UP
```

Now you can set the cluster as active to receive all the commands you issue with the *confluent* tool:

```
confluent kafka cluster use <Your Cluster ID>
```

__8. Enter the following command:

```
confluent kafka cluster
```

Confluent will print all the cluster-related sub-commands:

```
Usage:
  confluent kafka cluster [command]

Available Commands:
  create      Create a Kafka cluster.
  delete      Delete a Kafka cluster.
  describe    Describe a Kafka cluster.
  list        List Kafka clusters.
  update      Update a Kafka cluster.
  use         Make the Kafka cluster active for use in other commands.
```

The above output reveals the fact that there is no start/stop cluster command.

For Confluent CLI command reference, visit
https://docs.confluent.io/confluent-cli/current/command-reference/index.html

Note: You can avoid typing *confluent* at the command-line prompt for any CLI command by integrating the *confluent* prompt within the OS command prompt (essentially, starting the Confluent shell).

__9. Enter the following command:

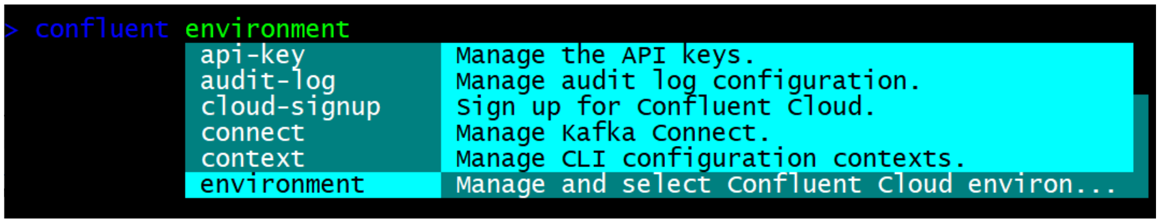
```
confluent shell
```

You should see the

> confluent

prompt appear under the OS command-line prompt; now you can type and execute the commands directly at the prompt.

Note that the command menu is limited to only 6 commands at a time; to navigate the menu, use the *Tab* and arrow keys.



```
> confluent environment
              api-key           Manage the API keys.
              audit-log         Manage audit log configuration.
              cloud-signup      Sign up for Confluent Cloud.
              connect           Manage Kafka Connect.
              context           Manage CLI configuration contexts.
              environment       Manage and select Confluent Cloud environ...
```

Spend some time understanding the *confluent* CLI environment.

__10. Finally find the exit command and hit enter. It will exit the shell.

Part 3 - Review

In this lab, you learned how to install and use the *confluent* CLI tool.

Lab 3 - Understanding Kafka Topics

A Kafka topic is a logical grouping of partitions, which are physical files on the file system where messages are published (written) to, stored, and consumed (read/fetched) from. Topics are partitioned across a cluster of machines for scalability and sustained throughput.

Kafka messages are key/value pairs where the key is commonly used for partitioning the events such that records (message values) with the same key are always written to the same topic partition.

Generally, topics are used by Kafka to categorize events/messages (we will use the *event* and *message* terms interchangeably) that have some sort of semantic similarity. When you write (publish messages) to a topic, the messages go to one of the topic's partitions depending on the key. Kafka uses a combination of the key's hash value and some load metrics to determine the partition the message will go to.

New messages are appended to a partition file, hence the reference to a partition as a transaction log file. Note that there are no random insert or message update capabilities in Kafka. Also note, that Kafka is not a queuing system of sorts as the queue abstraction implies message dequeuing (removing them from the queue) upon message consumption. Unlike many messaging systems, messages in Kafka topics are durable with options to be purged after a configurable TTL (time-to-live) period or when a preconfigured size has been reached.

Messages that are consumed from a topic are scanned (think an SQL table scan) starting from a location on a partition (controlled by the offset parameter) all the way to the partition's tail (which can be a moving target with new messages being constantly published to the topic and appended to the partitions), or from a message timestamp -- there is no SQL-type WHERE or BETWEEN message selection constructs or similar filtering capabilities. Kafka's primary use case is to act as a middleware component that acts as a high-performance scalable message bus connecting message publishers and message consumers in a simple and efficient message passing loop.

In this lab, you will learn how to create and configure topics in Confluent Cloud. Note that in one part of the lab, you will be required to use the *confluent* CLI tool you installed in one of the previous labs.

Note: The Confluent Cloud Web UI is constantly evolving and despite our best effort to keep up with the changes, you may not see certain UI elements as they were captured at the time of this writing.

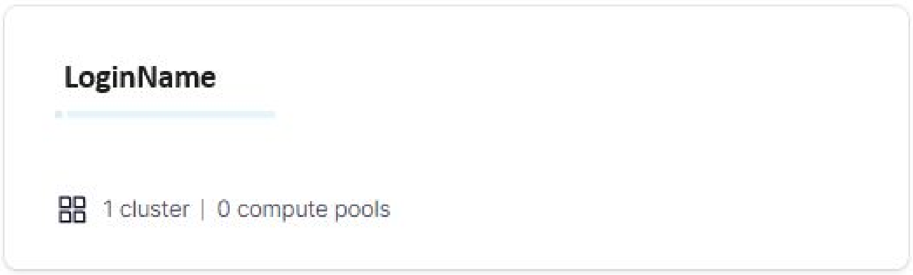
Part 1 - Log in to Confluent Cloud

__1. If you are signed out, log in back to Confluent Cloud at

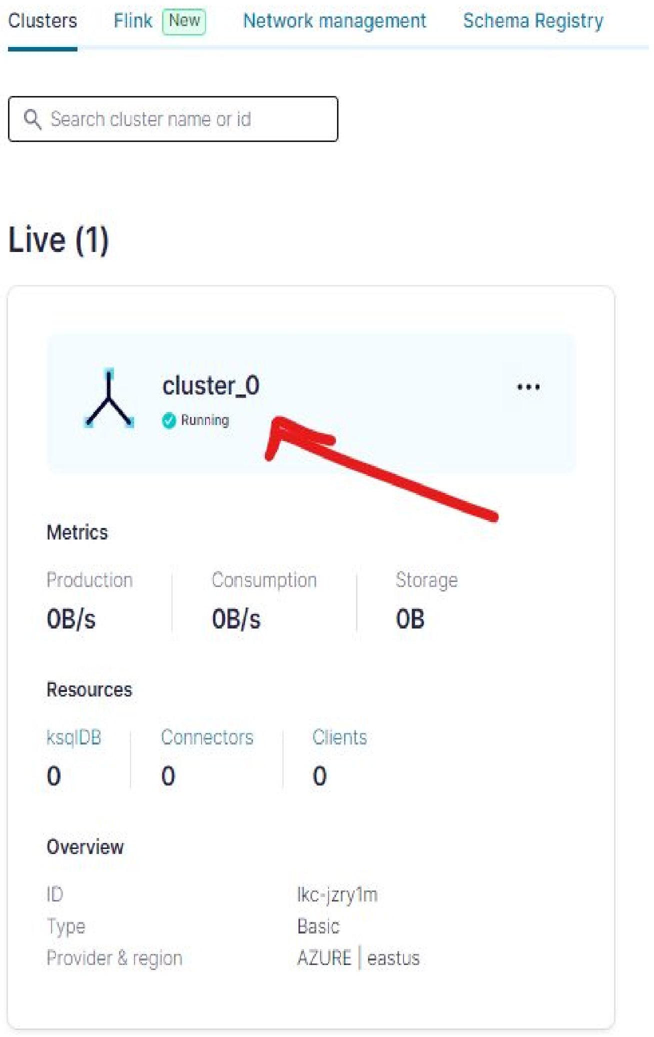
__2. https://confluent.cloud/

__3. Click the left hand menu item: Environments

__4. Then click on your "LoginName" as shown. It will be your username here.

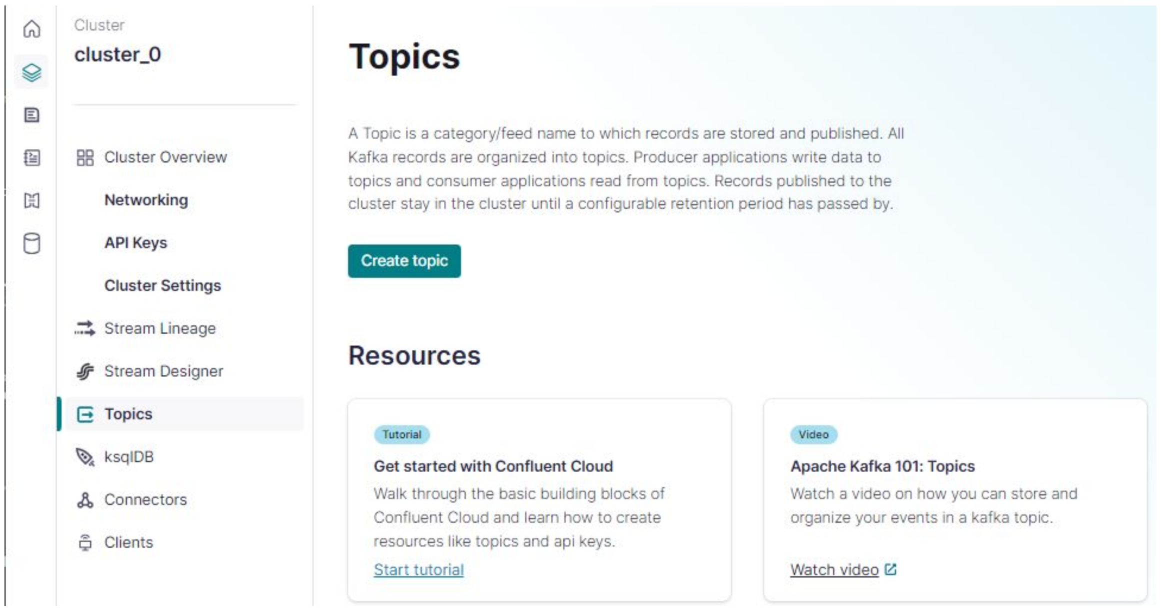


__5. Next, click your cluster name to see the Cluster Overview page for your cluster.



`Remember to select the cluster name you created in the earlier Lab`

__6. Choose **Topics** in the left-hand side navigation bar to see the Topics area.



Topics

A Topic is a category/feed name to which records are stored and published. All Kafka records are organized into topics. Producer applications write data to topics and consumer applications read from topics. Records published to the cluster stay in the cluster until a configurable retention period has passed by.

Create topic

Resources

Tutorial
Get started with Confluent Cloud
Walk through the basic building blocks of Confluent Cloud and learn how to create resources like topics and api keys.
Start tutorial

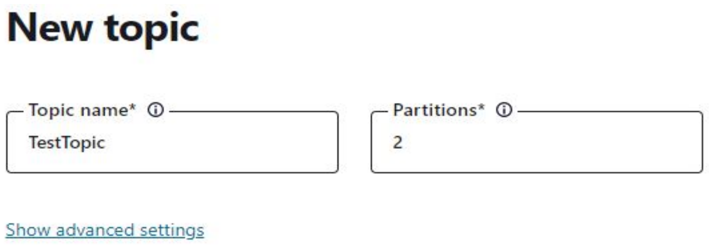
Video
Apache Kafka 101: Topics
Watch a video on how you can store and organize your events in a kafka topic.
Watch video

Part 2 - Create a Topic in the Confluent Cloud UI

__1. Click **Create topic** on the **Topics** page and create a topic with the following configuration settings in the new window:

* For *Topic* name use **TestTopic**

* For **Partitions**, enter **2**



New topic

Topic name*: TestTopic

Partitions*: 2

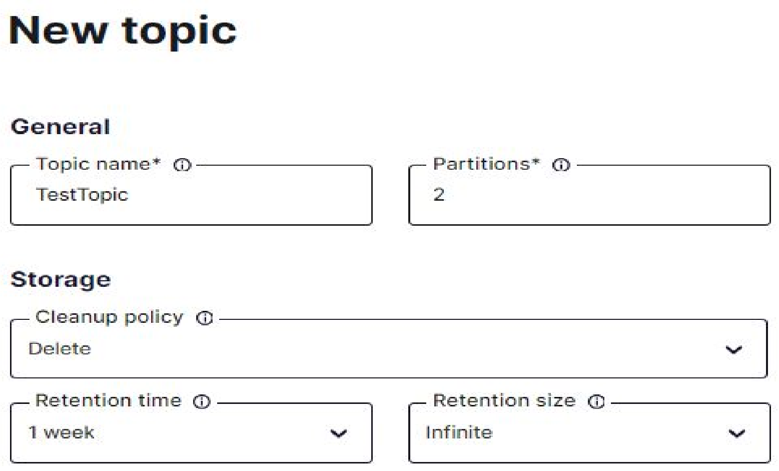
Show advanced settings

We have chosen two partitions to simplify the lab setup and help you focus on the learning process.

Note: For information on how to find the optimum number of topic partitions for your workloads, visit https://www.confluent.io/blog/how-choose-number-topics-partitions-kafka-cluster/

__2. Click **Show advanced settings**

Quickly review the main configuration settings available there.


New topic

General

Topic name*: TestTopic

Partitions*: 2

Storage

Cleanup policy: Delete

Retention time: 1 week

Retention size: Infinite

Note: The padlock icon next to a setting means that the setting is read-only.

One setting worth mentioning is replication.factor

Which is set to 3 (the value is locked in the Confluent Cloud environment -- you cannot edit it).

This configuration setting controls how many times each partition is replicated. Replication happens on different brokers, meaning that there will be three brokers involved.

The other setting to pay attention to is the min.insync.replicas

parameter, which is set to 2 and is locked as well.

In Kafka, data from producers is considered committed only when it has been written to all in-sync replicas and consumers can only see and read committed messages.

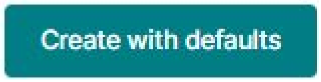
The replication.factor and min.insync.replicas configuration parameters are Kafka's data storage reliability guarantee as replicas are essentially data backups. When the number of replicas falls below min.insync.replicas, the brokers will stop accepting messages and start sending to the producers instances of NotEnoughReplicasException.

Basically, replication.factor (and, by extension, min.insync.replicas) is a dial between data consistency/durability (the higher the parameter, the more replicas (backups) you have) and data availability/latency(writing to replicas is a synchronous operation and incurs a disk IO cost).

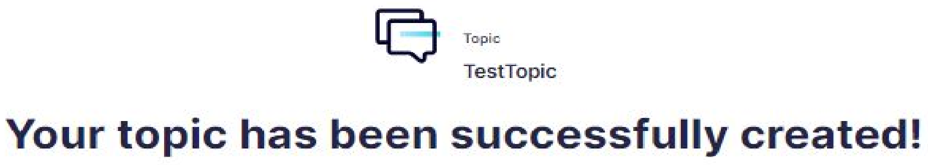
We are accepting the default values for all the settings (since practically all of them are locked down, anyway).

__3. Click **Use topic defaults** at the bottom of the page.

__4. Click **Create with defaults** at the bottom of the page.



You should see a message with your topic name shown.



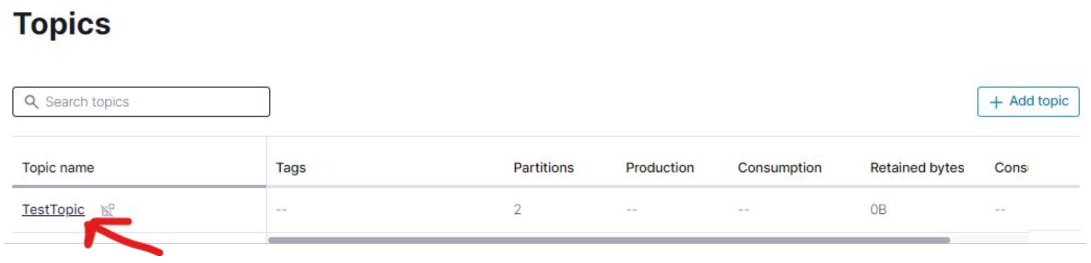
Part 3 - Understand Main Topic Concepts and Operations

We will work through the main topic-related operations for producing and searching for messages and introduce the main topic-related concepts.

First, we are going to produce some messages that will be placed on the topic we just created.

Note: Technically, messages are first sent to a broker process, which is part of the Kafka cluster, before being stored in a topic's partition. More specifically, producers communicate and write directly to the partition leaders of that topic.

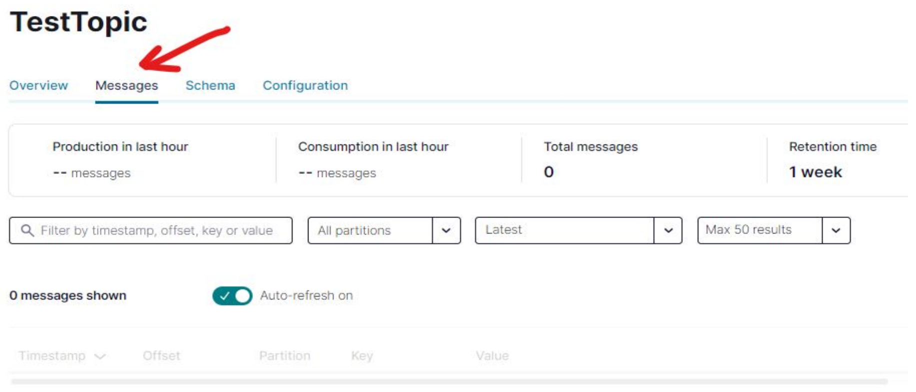
__1. On the left hand menu click the Topics link. You should see the TestTopic we just created.



Topic name	Tags	Partitions	Production	Consumption	Retained bytes	Cons
TestTopic	--	2	--	--	0B	--

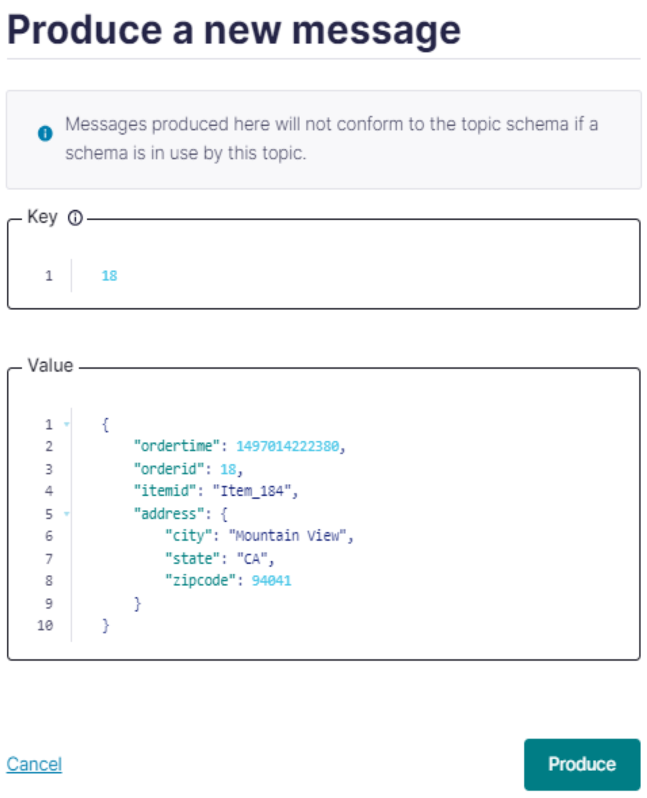
__2. Click your unique topic name, as shown above.

__3. Click the **Messages** tab as shown below.



__4. There are two ways to produce a message from here. Either select the **Produce new message** tab from the **Actions** drop down menu, *or* click the P**roduce new message** button.

__5. You will see a sample JSON-encoded message that you can use to test your topic.



Produce a new message

Messages produced here will not conform to the topic schema if a schema is in use by this topic.

Key

```
18
```

Value

```
{
    "ordertime": 1497014222380,
    "orderid": 18,
    "itemid": "Item_184",
    "address": {
        "city": "Mountain View",
        "state": "CA",
        "zipcode": 94041
    }
}
```

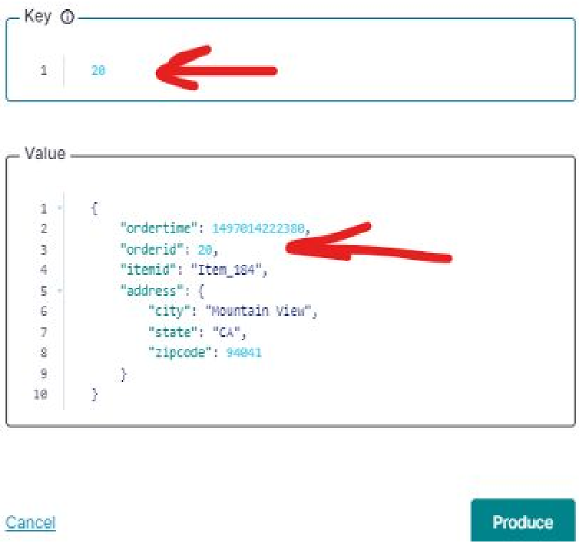
We are going to use the sample message with a small modification: we will change the **Key** property of the message to illustrate the placement of the messages on different partitions (we have 2).

You will recall that messages are key/value pairs where the key is commonly used for message partitioning across the topic where records (message values) with the same key are always written to the same partition.

__6. Make the following changes to the sample message:

* For the **orderid** field in the **Value** section of the JSON document, put **20**

* Enter **20** in the **Key** section

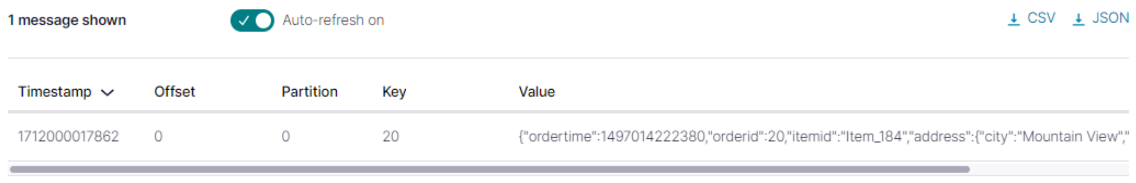


```
20
```

```
{
    "ordertime": 1497014222380,
    "orderid": 20,
    "itemid": "Item_184",
    "address": {
        "city": "Mountain View",
        "state": "CA",
        "zipcode": 94041
    }
}
```

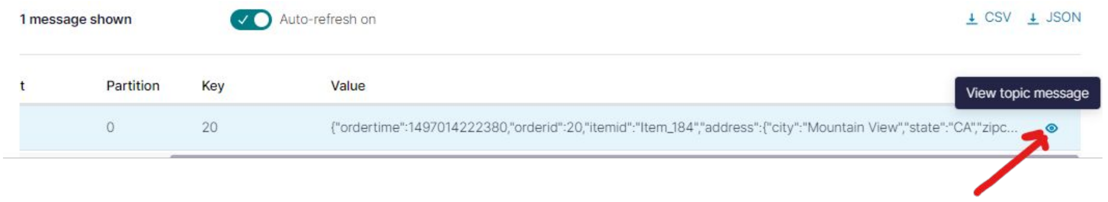
__7. Click **Produce**

After a moment, the message will be produced and sent to the topic.



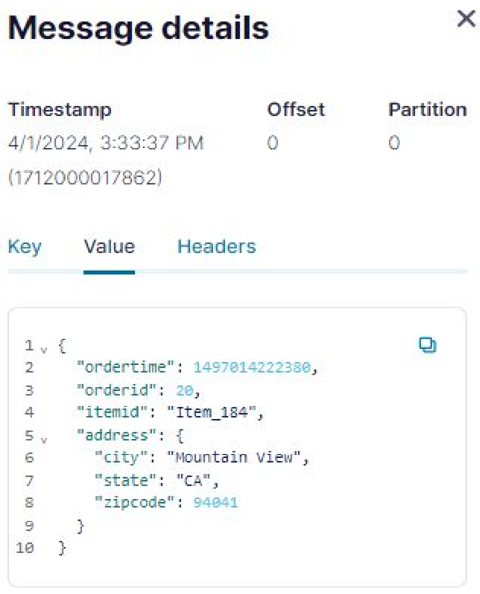
Timestamp	Offset	Partition	Key	Value
1712000017862	0	0	20	{"ordertime":1497014222380,"orderid":20,"itemid":"Item_184","address":{"city":"Mountain View","

Use the scroll bar to move to the right of the complete message as shown.



t	Partition	Key	Value
	0	20	{"ordertime":1497014222380,"orderid":20,"itemid":"Item_184","address":{"city":"Mountain View","state":"CA","zipc...

Message details fields listed, some of which are topic-specific, other are message-specific (the **key** and **value** fields) displayed as fields from the JSON document



4/1/2024, 3:33:37 PM (1712000017862) | 0 | 0

```
{
  "ordertime": 1497014222380,
  "orderid": 20,
  "itemid": "Item_184",
  "address": {
    "city": "Mountain View",
    "state": "CA",
    "zipcode": 94041
  }
}
```

The message itself is printed at the bottom of the page along with the Kafka system information: **Partition**, **Offset**, and **Timestamp**.

The *timestamp* message attribute is the Unix epoch time with a millisecond precision when the message was produced (as seen by the broker); it can help with monitoring the consumer lag metric. The consumer lag is a real-time metric for the time difference between the time when the message was produced and the time when it was consumed.

The first message on a partition is always placed with an offset 0 (at the head of the partition file). In our tests with the embedded producer, our first message was placed on the first topic partition (#0) -- it may not be the case for you as Kafka uses the **Key** part of the message to perform some sort of load balancing, which does not seem to be as simple as a key modulo operation, e.g. *key.hashCode % num_partions*. Nonetheless, Kafka will guarantee that messages with the same key will be routed to the same partition (birds of a feather (key) flock together).

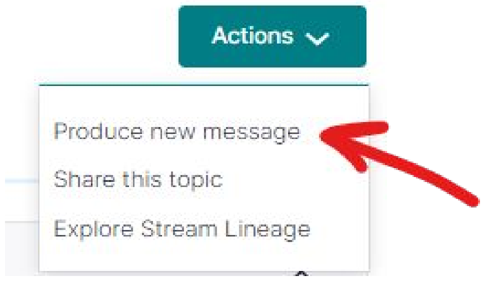
What is also important to note is that message ordering is guaranteed only within a single partition with no cross-partition order guarantee. In topics with more than one partition, the message ordering task, when and if needed, is the consumer's responsibility (caveat emptor!)

__8. Click the main TestTopic window are or use the **X** button on the message screen to exit the message topic window.

Part 4 - Adding additional messages

__1. Now we are going to produce some more messages by updating and publishing the sample message.

__2. Click the **Actions** button for your TestTopic, and select the Produce new message.

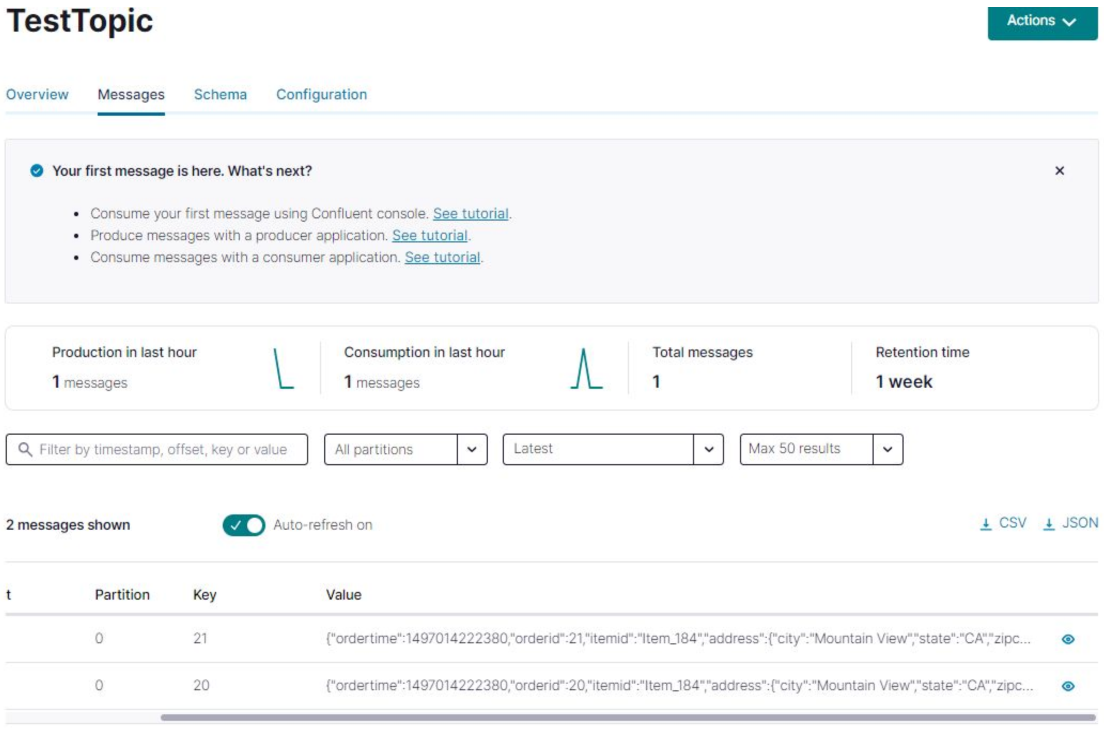


__3. Update the sample message as follows:

* For the **orderid** field in the **Value** section of the JSON document, put **21**

* Enter **21** in the **Key** section

__4. Click **Produce.** You should see two messages now on the main Topic window.



TestTopic

Overview | Messages | Schema | Configuration

Your first message is here. What's next?

- Consume your first message using Confluent console. See tutorial.
- Produce messages with a producer application. See tutorial.
- Consume messages with a consumer application. See tutorial.

Production in last hour	Consumption in last hour	Total messages	Retention time
1 messages	1 messages	1	1 week

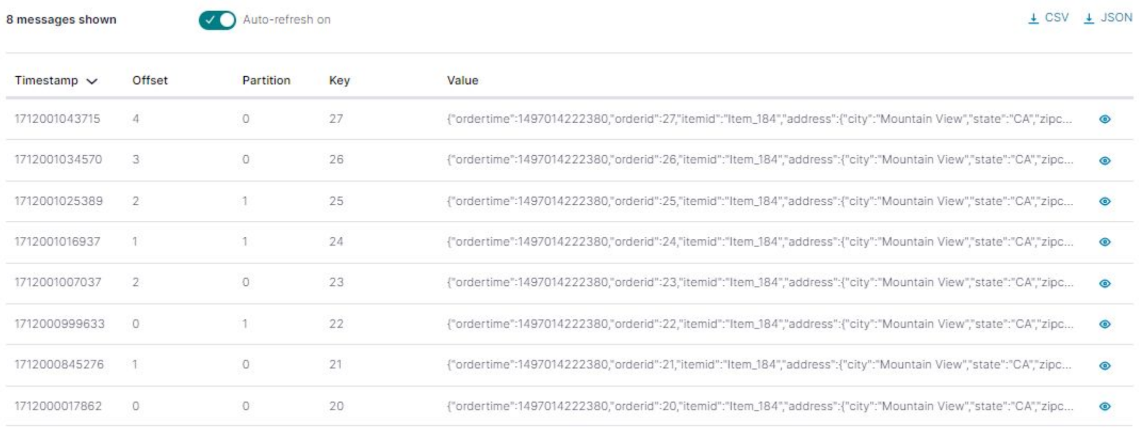
Filter by timestamp, offset, key or value | All partitions | Latest | Max 50 results

2 messages shown — Auto-refresh on — CSV — JSON

t	Partition	Key	Value
	0	21	{"ordertime":1497014222380,"orderid":21,"itemid":"Item_184","address":{"city":"Mountain View","state":"CA","zipc...
	0	20	{"ordertime":1497014222380,"orderid":20,"itemid":"Item_184","address":{"city":"Mountain View","state":"CA","zipc...

__5. Continue producing messages using auto-incremented **Key** and the **orderid** field values *up to 27* (using this sequence: 22, 23, 24, 25, 26, and 27) in order to achieve the goal of having a few messages on both partitions (Partition 0 and Partition1).

__6. In our tests, out of the 8 published messages, we ended up having 5 messages on partition 0 and 3 messages on partition 1 which indicates that Kafka does not employ a regular round-robin message distribution strategy.

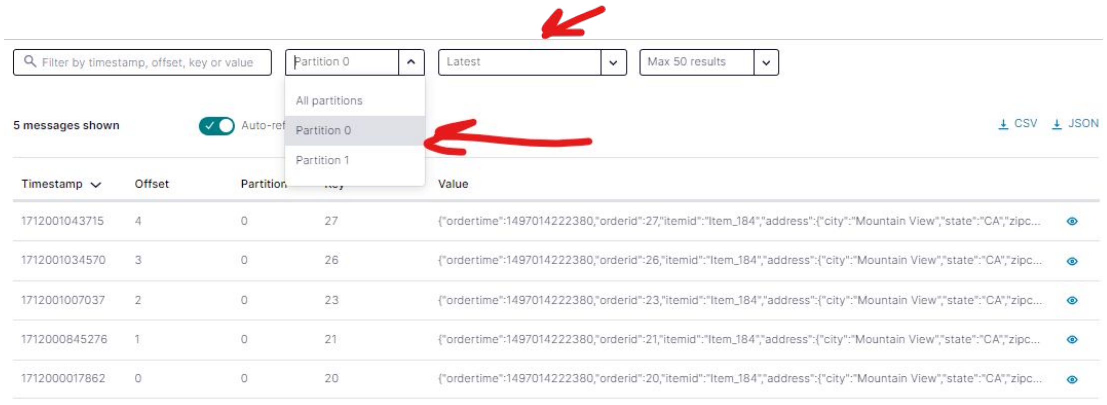


8 messages shown Auto-refresh on CSV JSON

Timestamp	Offset	Partition	Key	Value
1712001043715	4	0	27	{"ordertime":1497014222380,"orderid":27,"itemid":"Item_184","address":{"city":"Mountain View","state":"CA","zipc...
1712001034570	3	0	26	{"ordertime":1497014222380,"orderid":26,"itemid":"Item_184","address":{"city":"Mountain View","state":"CA","zipc...
1712001025389	2	1	25	{"ordertime":1497014222380,"orderid":25,"itemid":"Item_184","address":{"city":"Mountain View","state":"CA","zipc...
1712001016937	1	1	24	{"ordertime":1497014222380,"orderid":24,"itemid":"Item_184","address":{"city":"Mountain View","state":"CA","zipc...
1712001007037	2	0	23	{"ordertime":1497014222380,"orderid":23,"itemid":"Item_184","address":{"city":"Mountain View","state":"CA","zipc...
1712000999633	0	1	22	{"ordertime":1497014222380,"orderid":22,"itemid":"Item_184","address":{"city":"Mountain View","state":"CA","zipc...
1712000845276	1	0	21	{"ordertime":1497014222380,"orderid":21,"itemid":"Item_184","address":{"city":"Mountain View","state":"CA","zipc...
1712000017862	0	0	20	{"ordertime":1497014222380,"orderid":20,"itemid":"Item_184","address":{"city":"Mountain View","state":"CA","zipc...

Note: As you progress through your exercise, you are able to see the published messages, which won't be the case at a later time when the UI gets refreshed.

Now let's see how to search for the published messages using the search widget built right into the UI:



5 messages shown Auto-ref CSV JSON

Timestamp	Offset	Partition	Key	Value
1712001043715	4	0	27	{"ordertime":1497014222380,"orderid":27,"itemid":"Item_184","address":{"city":"Mountain View","state":"CA","zipc...
1712001034570	3	0	26	{"ordertime":1497014222380,"orderid":26,"itemid":"Item_184","address":{"city":"Mountain View","state":"CA","zipc...
1712001007037	2	0	23	{"ordertime":1497014222380,"orderid":23,"itemid":"Item_184","address":{"city":"Mountain View","state":"CA","zipc...
1712000845276	1	0	21	{"ordertime":1497014222380,"orderid":21,"itemid":"Item_184","address":{"city":"Mountain View","state":"CA","zipc...
1712000017862	0	0	20	{"ordertime":1497014222380,"orderid":20,"itemid":"Item_184","address":{"city":"Mountain View","state":"CA","zipc...

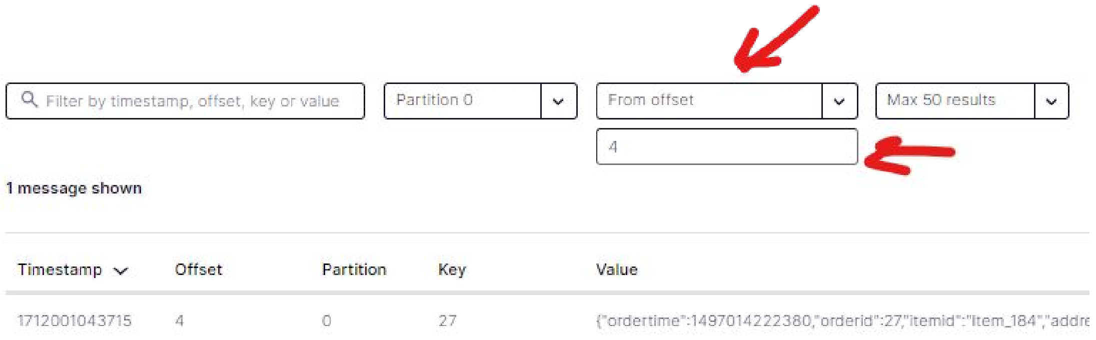
First, in the middle window, select your search option:

- In the Partition drop down choose, **Partition 0**,
- In the Partition drop down choose, **Partition 1**, or

Then, put in a (context-specific) value in the second window.

Let's see this in action.

__7. In the Latest drop down select **From offset** in the middle window and enter **4** in the second window.



4

1 message shown

Timestamp	Offset	Partition	Key	Value
1712001043715	4	0	27	{"ordertime":1497014222380,"orderid":27,"itemid":"Item_184","addre

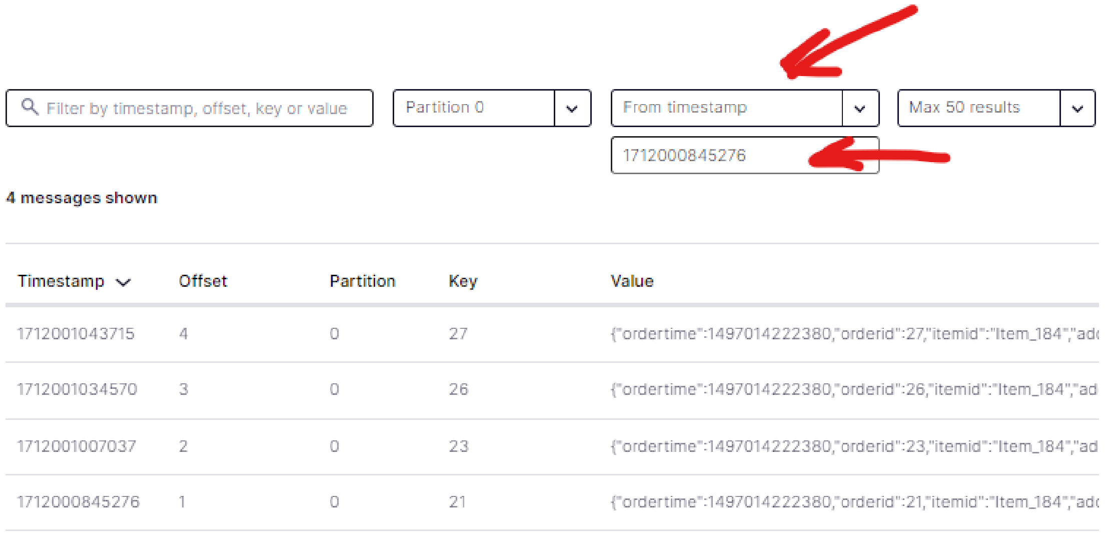
In our tests, in partition 0, we have one message from (and including) offset 4 (which is mapped to the last message on the partition) -- it may be different in your case

The Confluent Cloud built-in test consumer performs a partition scan from the requested offset down to the partition tail retrieving all the messages that are found.

__8. Click inside any of the messages that came up in response to your query and copy the *Timestamp* field's value.

__9. Select **From Timestamp** and enter the *Timestamp* field's value you just copied, then select the partition where that message was found (it was partition 0 in our case)

The Confluent Cloud built-in test consumer performed a partition scan from the requested timestamp retrieving all the messages on its way -- there were 4 messages in our case.



1712000845276

4 messages shown

Timestamp	Offset	Partition	Key	Value
1712001043715	4	0	27	{"ordertime":1497014222380,"orderid":27,"itemid":"Item_184","ad
1712001034570	3	0	26	{"ordertime":1497014222380,"orderid":26,"itemid":"Item_184","ad
1712001007037	2	0	23	{"ordertime":1497014222380,"orderid":23,"itemid":"Item_184","ad
1712000845276	1	0	21	{"ordertime":1497014222380,"orderid":21,"itemid":"Item_184","ad

Part 5 - View Topic Messages using the Confluent CLI

__1. Start or use the *confluent* CLI tool in a Command Prompt terminal.

__2. If you get a message that your session token has expired, enter the following command to re-login:

```
confluent login
```

__3. Enter the following command:

```
confluent kafka topic
```

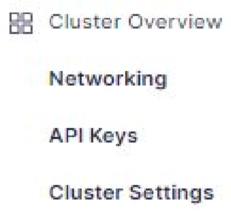
You should see the following output for the available *topic* sub-commands:

```
Available Commands:
  consume     Consume messages from a Kafka topic.
  create      Create a Kafka topic.
  delete      Delete a Kafka topic.
  describe    Describe a Kafka topic.
  list        List Kafka topics.
  produce     Produce messages to a Kafka topic.
  update      Update a Kafka topic.
```

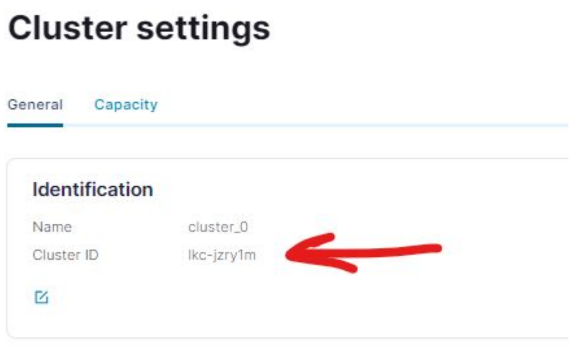
As you can see, with *confluent* you have full control over the lifecycle of a topic, including message consumption (with *confluent* acting as a message consumer).

Note that there is no "Edit" option as a topic, once created in Confluent Cloud, is considered a read-only resource.

__4. In the browser, click Cluster settings under Cluster overview.



__5. If you didn't create a txt file called clusterdetails.txt earlier, copy the cluster ID, you will use it in the next command. *Yours will be different than shown.*



Cluster settings

Identification

Name cluster_0

Cluster ID lkc-jzry1m

__6. Enter the following command to get **TestTopic**'s details:

```
confluent kafka topic describe TestTopic --cluster lkc-0xwo39
```

Replace the cluster ID [lkc-0xwo39] in this example with your own.

You should see the following --cluster output (abridged for space below):

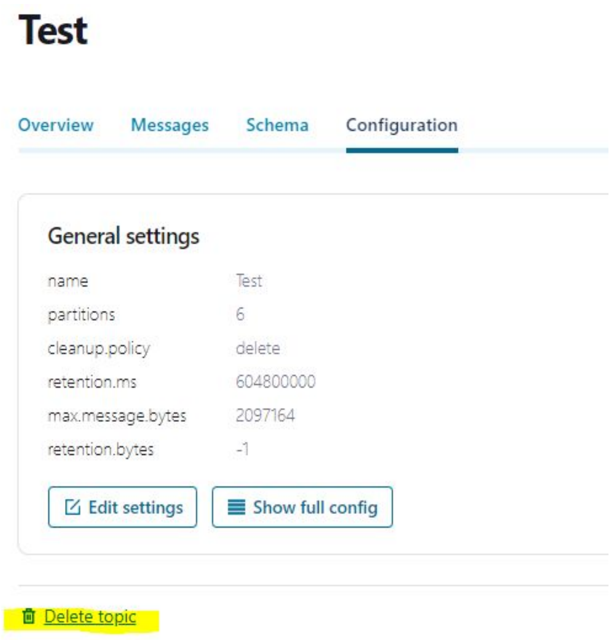
```
                   Name                   |        Value
------------------------------------------+----------------------
  cleanup.policy                          | delete
  compression.type                        | producer
  delete.retention.ms                     |             86400000
  file.delete.delay.ms                    |                60000
  flush.messages                          | 9223372036854775807
  flush.ms                                | 9223372036854775807
  follower.replication.throttled.replicas |
  index.interval.bytes                    |                 4096
  leader.replication.throttled.replicas   |
  max.compaction.lag.ms                   | 9223372036854775807
  max.message.bytes                       |                   30
  message.downconversion.enable           | true
  message.format.version                  | 3.0-IV1
. . .
```

Part 6 - Deleting a Topic (For Review Only)

You can delete a topic either through the **Topics** page in the Confluent Cloud portal or using the CLI.

Note: We are not going to delete your newly created **TestTopic** topic in this lab.

In the Confluent Cloud portal, under the **Configuration** tab for a topic you would see the **Delete topic** link at the bottom of the page, as shown below. Using this link will remove the topic, you must also confirm the deletion for it to happen.


Test

General settings

name	Test
partitions	6
cleanup.policy	delete
retention.ms	604800000
max.message.bytes	2097164
retention.bytes	-1

And here is the CLI command for deleting a topic:

```
confluent kafka topic delete <Your topic name>
```

This is the last step in this lab.

Part 7 - Review

In this lab, you learned how to create and interact with a topic in Confluent Cloud.

Lab 4 - Using the Confluent CLI to Consume Messages

In this lab, you will learn how to use the *confluent* CLI tool to provision an API key/secret credentials pair and consume messages.

This lab depends on the unique named **TestTopic** topic you created and populated with messages and the *confluent* CLI tool you installed locally in previous labs. If you deleted it, you must rerun that Lab.

Part 1 - Create an API Key and Consume Messages from Topic

__1. Start or use the *confluent* CLI tool for Confluent Cloud.

__2. Enter the following command if your session token has expired:

```
confluent login
```

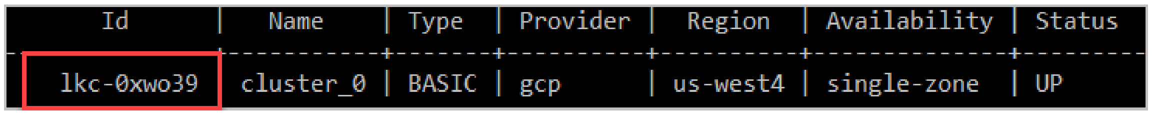
In order to consume messages from a topic, you need to have an API key for the cluster that hosts the topic. You can either use an existing API key or provision one on demand.

To provision an API key for a cluster, you need to know the cluster ID (you can look it up in the Confluent Cloud portal).

__3. To know your own ID execute the command:

```
confluent kafka cluster list
```

__4. Copy your cluster Id:



```
      Id      |   Name    | Type  | Provider |  Region  | Availability | Status
--------------+-----------+-------+----------+----------+--------------+---------
  lkc-0xwo39  | cluster_0 | BASIC | gcp      | us-west4 | single-zone  | UP
```

__5. Enter the following command as an API key provisioning request, supplying your cluster ID:

```
confluent api-key create --resource lkc-0xwo39
```

Replace lkc-0xwo39 with your own cluster Id.

You should see the following output listing the generated API key and the secret:

```
+---------+------------------------------------------------------------------+
| API Key | 3PFV.......AP                                                    |
| Secret  | JRqj8e9CFGZ......jrqtyfrpy5beF+Ft                                |
+---------+------------------------------------------------------------------+
```

__6. Enter the following to store you credentials. Change the resource to the one you found:

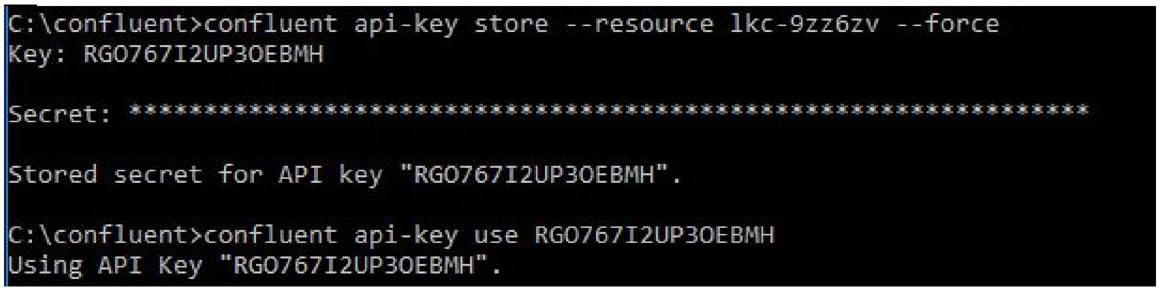
```
confluent api-key store --resource lkc-0xwo39 -force
```

__7. Fill in the Api Key and Secret when prompted.

__8. Enter the following replacing your api-key.

```
confluent api-key use <Your API key>
```

__9.



```
C:\confluent>confluent api-key store --resource lkc-9zz6zv --force
Key: RGO767I2UP3OEBMH

Secret: ****************************************************************

Stored secret for API key "RGO767I2UP3OEBMH".

C:\confluent>confluent api-key use RGO767I2UP3OEBMH
Using API Key "RGO767I2UP3OEBMH".
```

__10. Enter the following command specifying the offset and partition information as well as the API key/secret [in 1 line] and your unique cluster id name:

```
confluent kafka topic consume TestTopic --offset 2 --partition 0 --
cluster <Your cluster Id>
```

You should see the following output:

```
Starting Kafka Consumer. Use Ctrl-C to exit.
{"ordertime":1497014222380,"orderid":23,"itemid":"Item_184","address":{"city":"Mountain
View","state":"CA","zipcode":94041}}
{"ordertime":1497014222380,"orderid":26,"itemid":"Item_184","address":{"city":"Mountain
View","state":"CA","zipcode":94041}}
{"ordertime":1497014222380,"orderid":27,"itemid":"Item_184","address":{"city":"Mountain
View","state":"CA","zipcode":94041}}
```

Review the output.

__11. Press **Ctrl-C** to stop the local consumer.

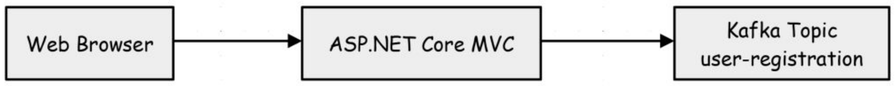
Part 2 - Review

In this lab, you learned how to use the *confluent* CLI tool to provision an API key/secret credentials pair and consume messages from a topic.

Lab 5 - Creating an ASP.NET Core MVC Kafka Client

In this lab, you will add Kafka support to an existing .NET 6 ASP.NET Core MVC application. The website will allow user registration. Each user registration request will be sent to a Kafka topic named *user-registration.*

The overall architecture looks like this:



NOTE: If the dotnet build or dotnet run command does not connect to the NuGet repository, then either way for 15-20 mins for the throttling to get removed OR students can copy the dependencies from the solutions file.

Part 1 - Explore an existing ASP.NET Core MVC web frontend

In this part, you will explore an existing ASP.NET Core MVC application. The UI and the essential controller logic is already implemented. You will add the code needed to integrate the application with the Kafka cluster.

__1. Using File Explorer, navigate to *C:\LabFiles*.

__2. Extract kafka-aspnet-frontend.zip under *C:\LabWorks*.

Note: If the *C:\LabWorks* directory doesn't exist, create it before copying the directory.

__3. Open Visual Studio Code (VSCode).

Note: The lab instructions will use Visual Studio Code. You can also choose to use Visual Studio if it's available to you.

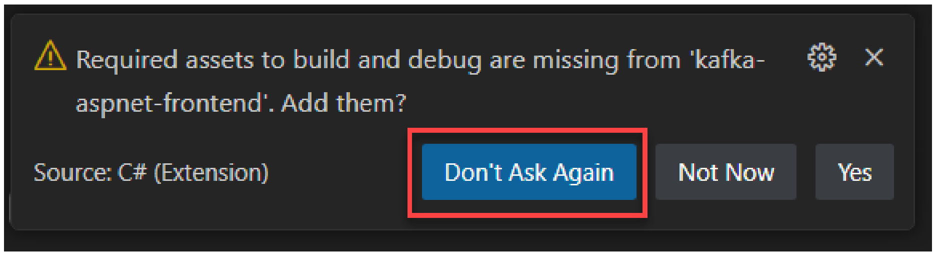
__4. In VSCode, click File | Auto Save.

This will ensure the files are automatically saved when changes are made to them.

__5. In VSCode, click File | Open Folder and select the *C:\LabWorks\kafka-aspnet-frontend* directory.

__6. You may need to click **Yes, I trust the authors** to continue.

__7. Select **Don't Ask Again** if you see the below dialog in the lower right corner of VSCode.



Required assets to build and debug are missing from 'kafka-aspnet-frontend'. Add them?

Source: C# (Extension)

__8. On the menu bar, click **Terminal** | **New Terminal**.

__9. In the terminal window, execute the following command to build the sample application:

```
dotnet build
```

Note: If the dotnet build command fails, execute the following command to add the NuGet repo location.

```
dotnet nuget add source https://api.nuget.org/v3/index.json -n
"NuGet.org"
dotnet clean
```

After the above commands, try the following command again:

```
dotnet build
```

__10. After the build process completes, execute the following command to run the ASP.NET MVC application in the built-in webserver:

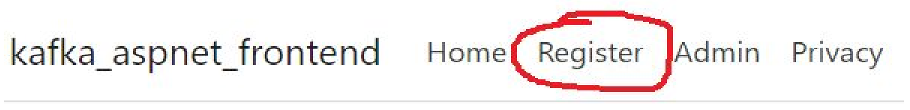
```
dotnet run --urls=http://localhost:8080
```

__11. Open the Chrome browser. If Chrome is not available, you can use Microsoft Edge.

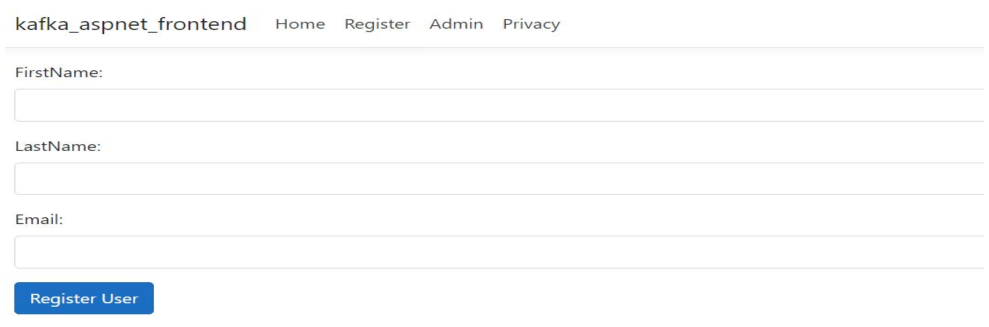
__12. In the browser, navigate to the following URL:

http://localhost:8080

__13. On the menu bar, click **Register**.



Notice the site looks like this:



The site isn't operational yet. You will add the code later in the lab.

__14. In the VSCode terminal window, press **Ctrl+C** to stop the built-in webserver.

Take a moment to review code in the following files:

1. **`Models\UserRegistrationModel.cs`** *(This file contains the model for the User Registration form)*

2. **`Controllers\HomeControllers.cs`** *(This file is already customized for the user registration form. It contains the GET & POST actions for the Register User button.)*

3. **`Views\Home\Register.cshtml`** *(This file contains the UI for user registration form.)*

Part 2 - Create a Kafka Topic

In this part, you will create a Kafka topic named *user-registration* that will be utilized from an ASP.NET Core MVC application.

__1. Open a Command Prompt and change directory to *<Your Local Dir>\confluent\*

Note: <Your Local Dir> is the location where you extracted the confluent tool as part of the Confluent Cloud CLI lab.

__2. Log on to Confluent Cloud by executing the following command:

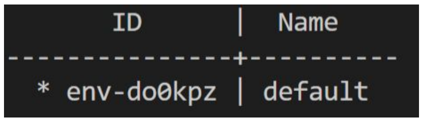
```
confluent login --save
```

Enter your credentials if prompted.

__3. Execute the following command to list environments available on Confluent Cloud:

```
confluent environment list
```

__4. Make a note of ID assigned to the default environment. Here's the sample ID assigned to the environment.



```
      ID       |   Name
---------------+----------
  * env-do0kpz | default
```

__5. Execute the following command to use the default environment:

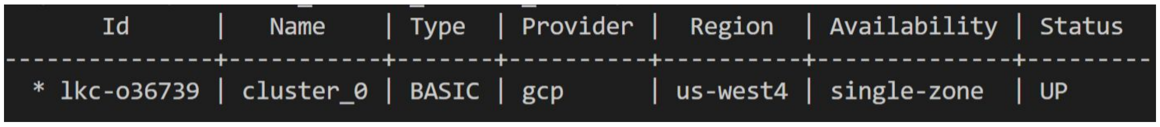
```
confluent environment use <Your Environment ID>
```

Note: Don't forget to use the ID you obtained in the previous step.

__6. Execute the following command to list Kafka clusters available in the environment:

```
confluent kafka cluster list
```

Here's the sample output. Make a note of the Id value in the first column.



```
      Id       |    Name    | Type  | Provider |  Region  | Availability | Status
---------------+------------+-------+----------+----------+--------------+---------
  * lkc-o36739 | cluster_0  | BASIC | gcp      | us-west4 | single-zone  | UP
```

__7. Execute the following command to use the Kafka cluster whose Id value you noted in the previous step:

```
confluent kafka cluster use <Your Cluster ID>
```

__8. Execute the following command to create a topic named *user-registration* that you will use later in the ASP.NET Core project:

```
confluent kafka topic create user-registration
```

__9. Execute the following command to ensure the Kafka topic was successfully created:

```
confluent kafka topic list
```

Part 3 - Obtain the api-key and cluster endpoint

The ASP.NET Core MVC client application requires an api-key so it can connect to the Kafka cluster. In this part, you will generate the api-key and also obtain the cluster endpoint.

__1. Execute the following command to obtain the cluster list:

```
confluent kafka cluster list
```

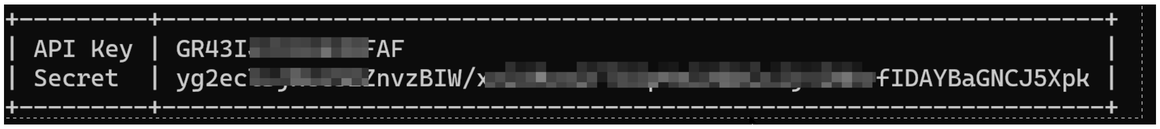
Make a note of the Id listed in the first column.

__2. Execute the following command to generate a new api-key and see it in a human-friendly way:

```
confluent api-key create --resource <Your Cluster Id> --output human
```

Note: Don’t forget to use the cluster Id you obtained in the previous step.

Notice the output looks like this.

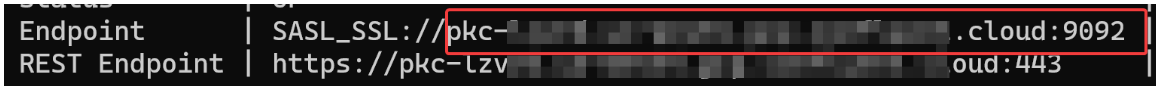


```
+---------+------------------------------------------------------------------+
| API Key | GR43I[redacted]FAF                                               |
| Secret  | yg2ec[redacted]ZnvzBIW/x[redacted]fIDAYBaGNCJ5Xpk                |
+---------+------------------------------------------------------------------+
```

__3. Save the API Key and Secret values in notepad since it will be utilized in various labs, including this one.

__4. Execute the following command to obtain the Kafka cluster endpoint:

```
confluent kafka cluster describe
```



```
 Endpoint      | SASL_SSL://pkc-[redacted].cloud:9092 |
 REST Endpoint | https://pkc-lzv[redacted]oud:443     |
```

__5. In notepad, save the Endpoint excluding the SASL_SSL:// prefix.

For example: *pad-levqd.us-west4.gcp.confluent.cloud:9092*

Make sure to save your Notepad, call it something like confluentcreds.txt

Part 4 - Add packages to the project

In this part, you will add packages required for working with Kafka to your project.

__1. In the terminal window of VSCode, execute the following commands one at a time to add the required packages to your project:

```
dotnet add package Confluent.Kafka -v 2.2.0

dotnet add package Newtonsoft.Json -v 13.0.3
```

Confluent.Kafka is required to add the Kafka integration to your project. It contains various classes that helps you to work with topics and to produce and consume messages.

Newtonsoft.Json lets you serialize objects to JSON and deserialize it from JSON back to the object. You will use this package later in the lab to serialize user registration data to JSON.

__2. In **VSCode** open **kafka-aspnet-frontend.csproj** file and notice the following Packages are added:

```
  <ItemGroup>
    <PackageReference Include="Confluent.Kafka" Version="2.2.0" />
    <PackageReference
Include="Microsoft.AspNetCore.Mvc.Razor.RuntimeCompilation"
Version="6.0.20" />
    <PackageReference Include="Newtonsoft.Json" Version="13.0.3" />
  </ItemGroup>
```

__3. Execute the following command to run the ASP.NET Core MVC application in the built-in webserver:

```
dotnet watch run --urls=http://localhost:8080
```

Note: Don’t forget to add the watch argument. The watch argument will be very handy later in the lab. You will make changes to the source code and the watch argument will automatically rebuild your application and re-host it in the built-in webserver.

Part 5 - Add the produce message functionality to the ASP.NET Core MVC application

In this part, you will implement the *Register User* functionality. The form contents will be converted to JSON and then write to the *user-registration* custom Kafka topic.

__1. In VSCode, open Controllers\HomeController.cs.

__2. Near the top of the file, below the existing namespaces, add the following namespaces:

```
using Confluent.Kafka;
using Newtonsoft.Json;
```

These namespaces contain classes that are required to integrate the application with Kafka and to manipulate JSON data. You will use various classes from these namespaces later in the lab.

__3. In the *HomeController* class, locate the comment
// TODO: deliveryHandler method and add the following code:

```
void deliveryHandler(DeliveryReport<string, string> deliveryReport)
{

    if (deliveryReport.Error.Code == ErrorCode.NoError)
    {
       Debug.WriteLine($"\n* Message delivered to:
({deliveryReport.TopicPartitionOffset}) with these details:");

       Debug.WriteLine($"-- Key: {deliveryReport.Key},\n-- Timestamp:
{deliveryReport.Timestamp.UnixTimestampMs}");

    }
    else
    {
        Debug.WriteLine($"Failed to deliver message with error:
{deliveryReport.Error.Reason}");
     }
}
```

This customer method will be used as a callback handler by your code that will produce messages in the user-registration Kafka topic. The **if** part will be called when there is no error and the **else** part will be called when there's an error while trying to send the message to Kafka.

Note: The Debug.WriteLine messages will only show up if you run the application in Debug mode.

__4. In the *HomeController* class, locate the comment // **TODO: Produce method** and add the following code:

```
void Produce(string topicName, string key, string value, ClientConfig
config)
{
  using (IProducer<string,string> producer = new
ProducerBuilder<string, string>(config).Build())
  {
    double flushTimeSec = 7.0;

    Message<string, string> message = new Message<string, string> { Key
= key, Value = value };

     producer.Produce(topicName, message, deliveryHandler);

     Debug.WriteLine($"Produced/published message with key:
{message.Key} and value: {message.Value}");

     var queueSize =
producer.Flush(TimeSpan.FromSeconds(flushTimeSec));
      if (queueSize > 0)
      {
          Debug.WriteLine($"WARNING: Producer event queue has not been
fully flushed after {flushTimeSec} seconds; {queueSize} events
pending.");
       }
   }
}
```

The ProduceBuilder class uses the Kafka configuration to create an IProducer object that uses the Produce method to write a message to the user-registration Kafak topic. The method also specifies a callback method so it can find it if the message was successfully written to the topic.

__5. Inside the *Register* POST method, locate the comment
// **TODO: client config - hard-coded** and add the following Kafka configuration code. (**Note**: Don't forget to replace Kafka cluster Endpoint, API key, and API secret with values you noted earlier in the lab which you saved in Notepad):

```
ClientConfig clientConfig = new ClientConfig();

clientConfig.BootstrapServers = "<Your Kafka Cluster Endpoint>";

clientConfig.SecurityProtocol = SecurityProtocol.SaslSsl;

clientConfig.SaslMechanism = SaslMechanism.Plain;

clientConfig.SaslUsername = "<Your Kafka API key>";

clientConfig.SaslPassword = "<Your API Kafka Secret>";
clientConfig.SslCaLocation = "probe";
```

The code uses the ClientConfig class provided by the Confluent Kafka package to configure the cluster endpoint and the API-key based credentials.

__6. Inside the *Register* POST method, locate the comment // **TODO: Register logic** and add the following code, replacing with your unique topic name:

```
string topicName = "user-registration";

string key = Guid.NewGuid().ToString();
model.UserId = key;
string message = JsonConvert.SerializeObject(model);

Produce(topicName, model.UserId, message, clientConfig);
```

The lines in bold specify *user-registration* topic name, generates a new GUID, assigns it to UserId, converts the contents of the form available in the model object to JSON, and calls the Produce helper function to write it to the Kafka topic.

TROUBLESHOOTING: Ensure your code in VSCode got compiled and the built-in webserver restarted by itself. Occasionally, the watch argument doesn't work. In that case, press Ctrl+C, and run the dotnet watch run urls=http://localhost:8080 command again.

Part 6 - Test out the Produce functionality

In this part, you will test the produce functionality. You will start the confluent CLI consumer and the ASP.NET Core MVC application that will write messages to the Kafka topic. For now, you will use the Confluent CLI consumer to view the messages.

__1. In the terminal where you ran the confluent CLI commands earlier in the lab, execute the following command to start the Confluent CLI consumer. (Note: It is a single command.):

```
confluent kafka topic consume user-registration --cluster <Your cluster
Id>
```

Note: Don't forget to use the values you noted for topic and cluster Id (NOT the endpoint).

Your API key, and API secret was saved earlier in Lab 1 in the Lab Part "Create an API

Key and Consume Messages from Topic". If this messages below does not return troubleshoot that section.

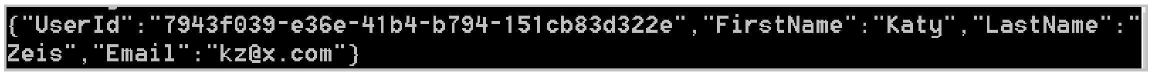
The following output should show up if the CLI consumer starts up successfully.
Starting Kafka Consumer. Use Ctrl-C to exit.

__2. Switch to the browser where you have **http://localhost:8080** open and click the ***Register*** menu item.

__3. Enter some values in the fields and click the **Register User** button.

```
FirstName: Katy
LastName: Zeis
Email : kz@x.com
```

__4. Switch to the Confluent CLI consumer and notice a message like this shows up: (Press enter on your keyboard if nothing is showing after a minute or so,)



```
{"UserId":"7943f039-e36e-41b4-b794-151cb83d322e","FirstName":"Katy","LastName":"
Zeis","Email":"kz@x.com"}
```

__5. Note: The values will vary depending on what you entered in the form.

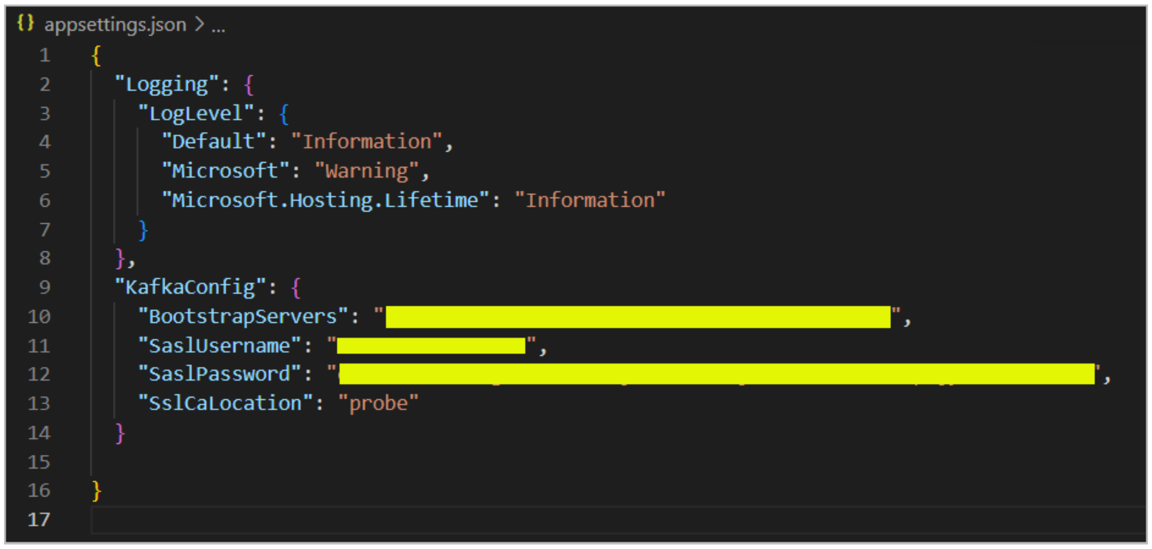
Part 7 - Move the Kafka configuration code to appsettings.json

__1. In VSCode, open **appsettings.json** and add the code after the closing "Logging" and before the last closing }. (Note: Don't forget to add the comma before adding "KafkaConfig":

```
,
"KafkaConfig": {
  "BootstrapServers": "<Your Cluster Endpoint>",
  "SaslUsername": "<Your API Key>",
  "SaslPassword": "<Your API secret>",
  "SslCaLocation": "probe"
}
```

Don't forget to replace the cluster endpoint, API key & secret with values you noted in the previous parts of the lab.

It should look like this but with your values:



appsettings.json > ...

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  },
  "KafkaConfig": {
    "BootstrapServers": "[redacted]",
    "SaslUsername": "[redacted]",
    "SaslPassword": "[redacted]",
    "SslCaLocation": "probe"
  }

}
```

__2. In VSCode, open Models\KafkaConfigModel.cs

__3. In the class, below the // **TODO: kafka config** comment, add the following class:

```
public class KafkaConfigModel
{
  public string? BootstrapServers { get; set; }
  public string? SaslUsername { get; set; }
  public string? SaslPassword { get; set; }
  public string? SslCaLocation { get; set; }
}
```

Notice you are matching the property names to the properties that were used in the appsettings.json file.

__4. In **VSCode**, open ***Program.cs*** and locate the comment
// **TODO: map configuration file** in the *ConfigureServices* method.

__5. Below the comment add the following code to map the appsettings section to your custom class:

```
builder.Services.Configure<KafkaConfigModel>(builder.Configuration.GetS
ection("KafkaConfig"));
```

__6. In VSCode, open Controllers\HomeController.cs.

__7. Below the existing *using* statements on the top of the file, add the following using statement:

```
using Microsoft.Extensions.Options;
```

__8. In the *HomeController* class, locate the comment
// TODO: appsettings/dependency injection and add the code shown in bold:

```
private readonly ILogger<HomeController> _logger;

// TODO: appsettings/dependency injection
private readonly IOptions<KafkaConfigModel>? _appSettings;

public HomeController(ILogger<HomeController> logger,
IOptions<KafkaConfigModel> appSettings)
{
    _logger = logger;
    _appSettings = appSettings;
}
```

In these lines, the KafkaConfigModel class that is mapped to the appsettings.json section is injected in to the HomeController by using the dependency injection concept supported by ASP.NET Core MVC.

Make sure to add the comma as shown in yellow above

__9. In the ***HomeController*** class, locate the comment
// TODO: GetKafkaConfig method and add the following code:

```
// TODO: GetKafkaConfig method
ClientConfig GetKafkaConfig()
{
  ClientConfig clientConfig = new ClientConfig();
  clientConfig.BootstrapServers = _appSettings?.Value.BootstrapServers;
  clientConfig.SecurityProtocol = SecurityProtocol.SaslSsl;
  clientConfig.SaslMechanism = SaslMechanism.Plain;
  clientConfig.SaslUsername = _appSettings?.Value.SaslUsername;
  clientConfig.SaslPassword = _appSettings?.Value.SaslPassword;
  clientConfig.SslCaLocation = _appSettings?.Value.SslCaLocation;

  return clientConfig;
}
```

Notice the method constructs the ClientConfig object by reading values from the appsettings.json file.

__10. In the *Register* POST method, locate the comment
// TODO: client config - hard-coded and comment out the following statements:

```
// ClientConfig clientConfig = new ClientConfig();
// clientConfig.BootstrapServers = "<Your Kafka Cluster Endpoint>";
// clientConfig.SecurityProtocol = SecurityProtocol.SaslSsl;
// clientConfig.SaslMechanism = SaslMechanism.Plain;
// clientConfig.SaslUsername = "<Your Kafka API Key>";
// clientConfig.SaslPassword = "<Your Kafka API secret>";
// clientConfig.SslCaLocation = "probe";
```

__11. In the same *Register* method, locate the comment
// TODO: client config with appsettings/dependency injection and add the following code shown in bold:

```
// TODO: client config with appsettings/dependency injection
ClientConfig clientConfig = GetKafkaConfig();
```

Part 8 - Test the application

__1. Ensure there are no syntax errors and the code is auto-compiled by the watch argument. If you don't see the built-in webserver restarting in the VSCode terminal, press Ctrl+C in the terminal and run the command:

```
dotnet watch run --urls=http://localhost:8080
```

__2. Switch back to the browser where the *Register* page is open.

__3. Enter values in the fields and click the **Register User** button.

__4. Switch to the Confluent CLI consumer and ensure the record was successfully sent to the Kafka topic.

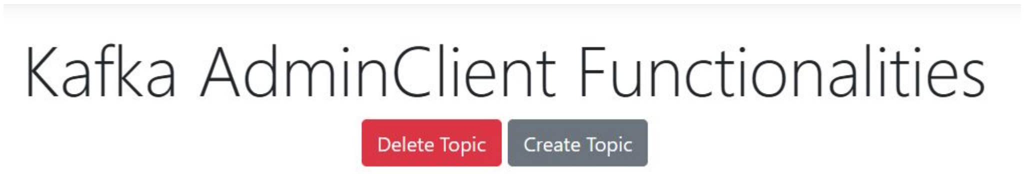
Part 9 - Add support for deleting a topic

__1. In this exercise, you will use the *AdminClient* capabilities to delete a Kafka topic.

__2. Switch to the browser where your ASP.NET MVC application is open.

__3. Click **Admin** in the menu bar.

Notice there are two buttons:



You will add the code for Delete Topic and Create Topic buttons later in this exercise.

__4. In VSCode, open **HomeController.cs**.

__5. Add the following statement towards the top of the file where namespaces are imported. (Note: The namespace includes various Kafka Admin oriented classes, such as AdminClientBuilder.)

```
using Confluent.Kafka.Admin;
```

__6. In HomeController.cs, locate the **DeleteTopic** method.

__7. Add the following code replacing the topic name with yours:

```
ClientConfig clientConfig = GetKafkaConfig();
var topicsToDelete = new List<string> { "user-registration" };

return View("Admin");
```

These lines load the Kafka configuration and define the topic you want to delete.

__8. Before the return statement, add the following code as shown in bold.

```
using (IAdminClient adminClient = new AdminClientBuilder(new
AdminClientConfig(clientConfig)).Build())
{

}

return View("Admin");
```

__9. Inside the using block, add the following code.

```
using (IAdminClient adminClient = new AdminClientBuilder(new
AdminClientConfig(clientConfig)).Build())
{
  try
  {
    await adminClient.DeleteTopicsAsync(topicsToDelete);
```

```
     ViewBag.Message = "Topic deleted!";
    }
    catch (Exception e)
    {
      ViewBag.Message = e.Message;
    }
}

return View("Admin");
```

When you use the await keyword, the compiler will complain that it's not enclosed in a function using async. You will fix the error in the next step.

__10. Modify the DeleteTopic method as shown in bold.

```
      public async Task<ActionResult> DeleteTopic()
```

__11. In VSCode, save HomeController.cs

Part 10 - Add support for creating a topic

In this exercise, you will use the *AdminClient* capabilities to create a Kafka topic.

__1. In HomeController.cs, locate the **CreateTopic** method.

__2. Add the following code shown in **bold** to the CreateTopic method. Remember to update with your unique topic name.

```
ClientConfig clientConfig = GetKafkaConfig();

var topicToCreate = new List<TopicSpecification> {
  new TopicSpecification { Name = "user-registration" }
};

return View("Admin");
```

The above statements import the Kafka configuration and define the topic that you want to create. TopicSpecification also lets you specify the number of partitions and other parameters.

__3. Before the return statement, add the following code as shown in **bold**.

```
using (IAdminClient adminClient = new AdminClientBuilder(new
AdminClientConfig(clientConfig)).Build())
{

}
```

```
return View("Admin");
```

__4. Inside the using block, add the following code shown in bold.

```
using (IAdminClient adminClient = new AdminClientBuilder(new
AdminClientConfig(clientConfig)).Build())
{
  try
  {
    await adminClient.CreateTopicsAsync(topicToCreate);
    ViewBag.Message = "Topic created!";
   }
   catch (Exception e)
   {
     ViewBag.Message = e.Message;
   }
}
```

When you use the await keyword, the compiler will complain that it's not enclosed in a function using async. You will fix the error in the next step.

__5. Modify the CreateTopic method as shown in **bold**.

```
public async Task<ActionResult> CreateTopic()
```

__6. In VSCode, save HomeController.cs

Part 11 - Test the DeleteTopic and CreateTopic functionalities

__1. Ensure there are no errors when you save the file and the dotnet watch automatically performs the build.

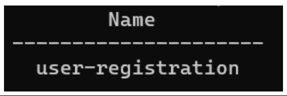
__2. Switch to the terminal window where you logged in into Confluent Kafka.

__3. Press CTRL+C to stop the Kafka consumer, if it's running.

__4. Execute the following command to verify the topic currently exists.

```
confluent kafka topic list
```

It should show the output like this with your unique topic name:



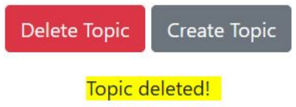
```
        Name
---------------------
  user-registration
```

__5. Switch to the browser window where your ASP.NET MVC application is running.

__6. Click **Admin** in the menu bar.

__7. Click Delete Topic.

__8. Ensure the *Topic deleted!* message shows up on the page.

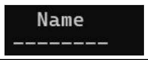


__9. Switch back to the terminal window where you have been executing the confluent commands.

__10. Execute the following command to get the topic list.

```
confluent kafka topic list
```

Ensure your user-registration topic is not listed.



```
  Name
---------
```

__11. Switch back to the browser and try clicking the *Delete Topic* button again.

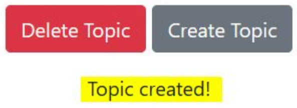
Notice the following error shows up:

An error occurred deleting topics: [user-registration]: [Broker: Unknown topic or partition].

This error is expected since the topic has already been deleted.

__12. Click the **Create Topic** button on the **Admin** page.

__13. Ensure Topic created! shows up on the page.



__14. Switch back to the terminal window where you have been running the confluent commands. Execute the following command to get the topic list.

```
confluent kafka topic list
```

__15. Ensure the user-registration topic is listed.

__16. Switch back to the browser and try clicking the *Create Topic* button again.

Notice the following error shows up: An error occurred creating topics: [user-registration]: [Topic 'user-registration' already exists.]. This error is expected since the topic already exists.

Result: You have successfully tested the topic deletion and creation functionality.

Part 12 - Clean-up

__1. In **VSCode** terminal, press **Ctrl+C** to stop the built-in webserver.

__2. If you have any process running, press **Ctrl+C**

__3. Keep the terminal window and **VSCode** running for the next lab.

Part 13 - Review

In this lab, you added Kafka support to an existing ASP.NET Core MVC application.

Lab 6 - Creating an ASP.NET Core WebAPI Kafka Client

In the previous lab, the MVC project directly sent a message to the *user-registration* Kafka topic. In this lab, you will decouple the MVC project from Kafka by introducing the WebAPI layer. You will add code to an existing WebAPI so it receives requests from the MVC application and writes messages to the *user-registration* Kafka topic.

The overall architecture looks like this:

This lab builds on top of the previous lab. At the very least, you must have a topic named *user-registration* in your Kafka cluster. You should also have the Kafka cluster Id, endpoint, API key, and API secret to work on this lab.

Part 1 - Setup

__1. Using File Explorer, navigate to *C:\LabFiles* and extract ***kafka-webapi-starter*** under **C:\LabWorks**.

__2. After extraction, ensure the following folders exist.

```
C:\LabWorks\kafka-webapi-starter\kafka-aspnet-frontend
C:\LabWorks\kafka-webapi-starter\kafka-webapi-backend
```

__3. Launch **VSCode**.

__4. Click File | Open Folder and select *C:\LabWorks\kafka-webapi-starter.*

__5. Ensure the following two folders show up in **VSCode** Explorer pane.

Part 2 - Explore the Existing WebAPI Project

In this part, you will explore the existing WebAPI project. You will add Kafka support to the project later in the lab.

__1. In VSCode, right click kafka-webapi-backend and select Open in Integrated Terminal.

__2. In the terminal window, execute the following command to build and host the WebAPI in the built-in web server.

```
    dotnet watch run --urls=http://localhost:9090
```

__3. Note: Ensure you use port number 9090 since it has been hard-coded in the frontend application.

__4. Open **Postman** using the desktop shortcut.

__5. In **Postman**, click the + to open a new tab.

__6. Ensure the method is set to **GET**.

__7. Change URL to

```
http://localhost:9090/api/user-registration
```

__8. Click **Send**.

__9. Ensure it shows the User Registration Service is available! in response body.

__10. In **Postman**, change method to **POST**

__11. Click the **Body** tab.

__12. Select the **raw** radio button.

__13. Click the drop down and select **JSON**

__14. Enter the following **JSON** fragment. (Note: Don't forget to enter your name.)

```
{
  "FirstName": "<Your Name Here>"
}
```

__15. Ensure your configuration looks like this:

__16. Click **Send.**

__17. Ensure the following message is displayed in the **Body** section:

```
Hello, <Your Name>
```

__18. Close **Postman** and switch back to **VSCode.**

__19. In VSCode, open

kafka-webapi-backend\kafka-webapi-backend.csproj.

__20. Notice the started project already has the following packages added to it.

```
<PackageReference Include="Confluent.Kafka" Version="2.2.0" />
<PackageReference Include="Newtonsoft.Json" Version="13.0.3" />
```

Note: You learned about these packages in the previous lab and added them to the project. Previously, these were in the MVC project. This time, these are in the WebAPI project.

Part 3 - Update the ASP.Net MVC file.

__1. In VSCode, open kafka-webapi-backend\Models\UserRegistrationModel.cs. It is the same file you used before.

__2. Open kafka-webapi-backend\Controllers\UserRegistrationController.cs.

__3. Review the Route annotation configured at the class level.

```
[ApiController]
[Route("/api/user-registration")]
public class UserRegistrationController : ControllerBase
{
```

__4. Review the *Get* and *Post* methods defined in the API Controller. The code is shown below for reference. This is the standard WebAPI programming technique that doesn't involve any Kafka.

```
    [HttpGet]
    public string Get()
    {
        return "User Registration service is available!";
    }

    [HttpPost]
    public string Post([FromBody] UserRegistrationModel body)
    {
        return $"Hello, {body.FirstName}";
    }
```

You will make changes to the Post method later in the lab so it can write a message to the custom user-registration Kafka topic.

Part 4 - Explore the ASP.NET Core MVC Frontend Application

In this part, you will explore the existing front-end application. In the previous lab, you had the Kafka client implemented in the MVC project. This time around, the MVC application will send HTTP requests to the WebAPI and that will be responsible for write messages to the Kafka topic.

__1. In VSCode, open kafka-aspnet-frontend\Controllers\HomeController.cs.

__2. Examine the *Register* POST method. It's using a custom helper method named *SendData*. The *SendData* is using the **HttpClient** class to post contents of the form stored in *UserRegistrationModel* based object to the WebAPI. It is also using asynchronous programming technique by using **Task**, **async**, **await**, **PostAsync**, and **ReadAsStringAsync** methods.

In this lab, you will not make any changes to the MVC front-end application since the Web API will be responsible for connecting to Kafka.

__3. In VSCode, right click *kafka-aspnet-frontend* and select Open in Integrated Terminal.

__4. Execute the following command in the terminal to launch the built-in web server and host the MVC application.

```
dotnet watch run --urls=http://localhost:8080
```

__5. Open Chrome and navigate to the following URL:

```
http://localhost:8080
```

__6. Click **Register.**

__7. Enter your name in the value in the *First Name* field and click **Register User**.

__8. Notice the Hello, <Your name> shows up.

So far you have tested the Web API and the MVC applications. The MVC application can send HTTP requests to the Web API project and receive the response from it.

Part 5 - Add Kafka Configuration to the WebAPI Project

In this part, you will add Kafka configuration to the WebAPI project. In the previous lab, you saw the basics of appsettings.json and dependency injection. In this lab, you will use a more streamlined approach to add configuration data to the appsettings.json and then load it from your API controller.

__1. In VSCode, open kafka-webapi-backend\appsettings.json.

__2. Add the content shown in **bold**. Don't forget to add a comma before the custom code and replace the values with yours from Notepad:

```
  "AllowedHosts": "*",
  "ProducerConfig": {
    "BootstrapServers": "<Your Cluster Endpoint>",
    "SaslMechanism": "Plain",
    "SaslUsername": "<Your API Key>",
    "SaslPassword": "<Your API Secret>",
    "SecurityProtocol": "SaslSsl",
    "SslCaLocation": "probe"
    }
}
```

Notice you are configuring everything related to Kafka in the appsettings.json file in a custom section named *ProducerConfig*. In the previous lab, you kept *SaslMechanism* and *SecurityProtocol* in the .cs file. This is to show you both approaches where you want to keep a few things in .cs or put everything in the appsettings.json file.

__3. Open kafka-webapi-backend\Program.cs.

Notice there are some namespaces pre-configured for you in the starter project.

__4. Locate the comment
// **TODO: add kafka configuration code** and add the following code.

```
var producerConfig = new ProducerConfig();
builder.Configuration.Bind("ProducerConfig",producerConfig);
builder.Services.AddSingleton<ProducerConfig>(producerConfig);
```

ProducerConfig is a predefined class provided in the Confluent.Kafka namespace. It's bound to the custom appsettings.json section named ProducerConfig. The two names don't have to match. Your custom section name can be different than the configuration class name. After loading the configuration, it is registered as a singleton object so it can be made available to other controllers via dependency injection.

__5. Open kafka-webapi-backend\Controllers\UserRegistrationController.cs.

Notice a few namespaces have already been added for you to the starter project.

__6. In the *UserRegistrationController* class, locate the comment
// TODO: appsettings/dependency injection and add the following code.

```
        private readonly ProducerConfig? _config;
```

__7. Modify the constructor as shown in **bold**.

```
public UserRegistrationController(ProducerConfig config)
{
    _config = config;
}
```

You are done with the Kafka configuration portion. With this approach, you didn't have to create a custom class to store Kafka configuration properties. This is useful when you don't want to customize the properties by adding additional properties.

Ensure there are no syntax errors in the terminal window where the dotnet watch command is running.

Part 6 - Add a Kafka Helper Class to the Project

In this part, you will add a Kafka helper class to the project that will let you produce/write messages to a Kafka topic.

__1. Right click ***kafka-webapi-backend*** and create a new folder named **Helpers**

__2. Using File Explorer, navigate to *C:\LabFiles\kafka-webapi-snippets* and drag & drop the ***ProducerHelper.cs*** file to VSCode *Helpers* folder.

Take a moment to study the ProducerHelper class. It is a class that will let you pass it the Kafka configuration, topic name, and the message you want to write to the Kafka topic.

__3. Ensure there are no build errors before moving on to the next step.

TROUBLESHOOTING: When you add a new file to the project by dragging & dropping it, often the dotnet watch command doesn't automatically trigger the build process. You can press Ctrl+C and then run the dotnet watch command that you used previously to force-build the project.

__4. Open kafka-webapi-backend\Controllers\UserRegisrationController.cs file.

__5. Near the top of the file, locate the comment
// TODO: add namespaces here and add the following namespace:

```
using kafka_webapi_backend.Helpers;
```

__6. In the ***UserRegistrationController.cs*** file, locate the Post method and change it as shown in **bold** below.

```
[HttpPost]
```

```
public async Task<string> Post([FromBody] UserRegistrationModel body)
{
  // TODO: Post method logic

  string topicName = "user-registration";

  string key = Guid.NewGuid().ToString();
  body.UserId = key;
  var response = "";
  try
  {
    string message = JsonConvert.SerializeObject(body);

    var producer = new ProducerHelper(this._config, topicName);

    await producer.SendMessage(key, message);

     response = $"Message written: {message}";
   }
   catch(Exception ex)
   {
     response = ex.Message;
   }

    return response;
}
```

The code uses the custom ProducerHelper class to write a message to the custom Kafka topic named user-registration.

__7. Ensure there are no build errors before moving on the to the next part.

Part 7 - Test the application

__1. Switch to the terminal window where you have Confluent CLI open.

__2. Execute the following command to log in.

```
confluent login --save
```

__3. Execute the following command to start the Confluent CLI consumer.

```
confluent kafka topic consume user-registration --cluster <Your Cluster
Id>
```

__4. Switch to Chrome where the ASP.NET MVC frontend application is open (Note: The URL should be http://localhost:8080).

__5. Click the **Register** menu item.

__6. Enter values in to the fields and click **Register User. After a moment you should see** a message below

__7. Switch to the terminal where the Confluent CLI consumer is running and ensure a message shows up in the terminal.

Also, check the terminal window where the kafka-webapi-backend is running and notice there is a message like this:

The message is using this format:

```
KAFKA => Delivered '{"UserId":"<Random GUID>","FirstName":"<Your first 
name>","LastName":"<Your last name>","Email":"<Your email>"}' to 'user-
registration [[1]] @3'
```

The last bit shows the topic name and the partition offset where the message was written.

Part 8 - Clean-up

__1. Press **Ctrl+C** in all terminal windows to stop the Confluent CLI, *kafka-webapi-backend, and kafka-aspnet-frontend.*

__2. *Keep the Confluent CLI terminal and **VSCode** open for the next lab.*

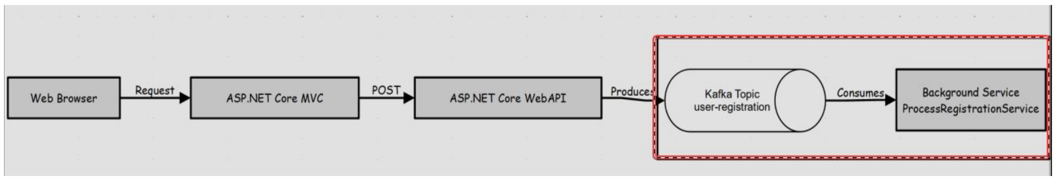
Part 9 - Review

In this lab, you added Kafka support to an existing WebAPI project and called it from the MVC project.

Lab 7 - Creating a .NET 6 Worker Kafka Client

In this lab, you will create a Web API hosted Worker (Background Service) that will continuously run and consume messages from the user-registration Kafka topic.

In the overall architecture, you will implement the portion highlighted in red.

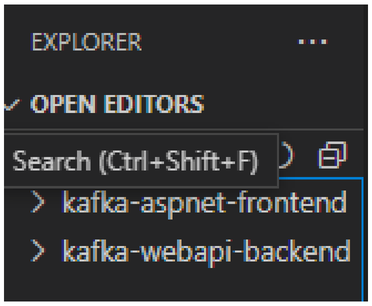


Part 1 - Set-up

__1. Ensure you have VSCode open with the **C:\LabWorks\kafka-webapi-starter** folder open from the previous lab.

NOTE: If you weren't able to complete the previous lab, you can use the solution for the previous lab from the C:\LabFiles\Solutions folder as the starter project for this lab. Don't forget to configure kafka-webapi-backend appsettings.json file with your Kafka endpoint, API key, and API secret.

__2. Ensure the following two folders show up in **VSCode** Explorer pane.



Part 2 - Create a .NET Web API Project

In this part, you will create a .NET Web API project and clean it up a bit. You will use this as the starter project to which you will add the .NET Core Worker.

__1. In VSCode EXPLORER pane, from the menu, click **Terminal → New Terminal**.

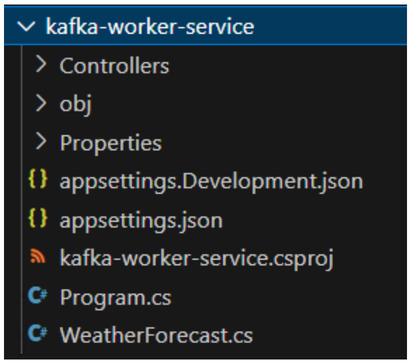
__2. Ensure the terminal shows the following prompt:

```
C:\LabWorks\kafka-webapi-starter>
```

__3. Execute the following command to create a new .NET Core Web API project:

```
dotnet new webapi --output kafka-worker-service
```

On the left, it will show in VSCode explorer.



```
kafka-worker-service
  Controllers
  obj
  Properties
  appsettings.Development.json
  appsettings.json
  kafka-worker-service.csproj
  Program.cs
  WeatherForecast.cs
```

__4. In the VSCode EXPLORER pane, expand **kafka-worker-service** and delete the *Controllers* folder.

__5. From the kafka-worker-service folder, delete WeatherForecast.cs

__6. In the terminal window opened in VSCode, execute the following command to switch to the kafka-worker-service project:

```
cd kafka-worker-service
```

__7. In the terminal window, execute the following command to add the package required to host a worker that can run in the background. You will create the worker in the next part of this lab:

```
dotnet add package Microsoft.Extensions.Hosting --version 6.0.1
```

__8. In VSCode, open **kafka-worker-service.csproj** and notice the package is available in the ItemGroup tag.

__9. Right click the **kafka-worker-service** project and create a folder named **Models**

__10. Using File Explorer navigate to *C:\LabFiles\kafka-worker-snippets* and drag & drop *UserRegistrationModel.cs* to VSCode kafka-worker-serivce\Models folder.

__11. Execute the following command to build the project and ensure there are no errors:

```
dotnet build
```

You have a base Web API project that you will use as the starting point for the .NET Core Worker project in the next lab.

Part 3 - Implement a .NET Core Worker

In this part, you will implement a .NET Core Worker that will be hosted in the Web API project you created in the previous part of the lab. A .NET hosted worker continuously runs in the background.

__1. In VSCode, right click **kafka-worker-service** and create a folder named **Services**

__2. Using File Explorer navigate to C:\LabFiles\kafka-worker-snippets and drag & drop UserRegistrationWorker.cs to VSCode kafka-worker-service\Services folder.

__3. In VSCode open UserRegistrationWorker.cs and review the code. Note: The file contents are listed below for reference:

```
public class UserRegistrationWorker : BackgroundService
    {
        // TODO: appsettings dependency injection

        public UserRegistrationWorker()
        {

        }

        protected override async Task ExecuteAsync(CancellationToken 
stoppingToken)
        {
            Console.WriteLine($"Worker started...");

    while (!stoppingToken.IsCancellationRequested)
            {
                // TODO: worker logic
                await Task.Delay(1000, stoppingToken);
            }
        }
    }
```

UserRegistrationWorker is a custom class that inherits from BackgroundService. The ExecuteAsync method is overriden in the custom class. In ExecuteAsync method uses a while loop to run continuously. You will later add the Kafka client code to the while loop so it can consume messages from the Kafka user-registration topic.

Now that you have a basic worker, without any Kafka specific code, let's register it so it can be hosted in the .NET Web API project and run continuously when you start up the

Web API project.

__4. In VSCode, open kafka-worker-service\Program.cs.

__5. Below the existing using statements, add the following using statement at the top of the file. It should be the first line of code:

```
using kafka_worker_service.Services;
```

__6. In Program.cs, locate this line (around line 10):

```
builder.Services.AddSwaggerGen();
```

__7. Add this line immediately after.

```
builder.Services.AddSingleton<IHostedService,
UserRegistrationWorker>();
```

Notice you are adding the custom UserRegistrationWorker (BackgroundService object) as a singleton object. This line ensures the worker gets registered/executed when the Web API is started/hosted.

__8. Execute the following command to build the project and ensure there are no syntax errors:

```
dotnet build
```

You have a .NET Web API hosted worker in place. It is not doing anything meaningful for now. You will add the Kafka consumer to it in the next part of the lab.

Part 4 - Add Kafka Support to the Worker Project

In this part, you will add Kafka packages and configuration to the worker project you created in the previous parts of this lab.

__1. In VSCode terminal window, execute the following command to add packages required to work with Kafka and data serialization/serialization:

```
dotnet add package Confluent.Kafka --version=2.2.0

dotnet add package Newtonsoft.Json --version=13.0.3
```

The worker will act as the Kafka consumer. It needs to have Kafka cluster configuration available to it. Let’s configure the appsettings.json file and add Kafka configuration to it.

__2. Using File Explorer, navigate to C:\LabFiles\kafka-worker-snippets and open appsettings-snippet.txt

__3. Copy the contents of the file to clipboard.

__4. Open kafka-worker-service\appsettings.json file and paste the contents as shown in bold. Note: Don't forget to add a comma before "ConsumerConfig". Also, replace the placeholders with values you used in the previous labs:

```
  "AllowedHosts": "*",
  "ConsumerConfig": {
    "BootstrapServers": "<Your Kafka Cluster Endpoint>",
    "SaslMechanism": "Plain",
    "SaslUsername": "<Your API Key>",
    "SaslPassword": "<Your API Secret>",
    "SecurityProtocol": "SaslSsl",
    "SslCaLocation": "probe",
    "GroupId": "kafka-lab"
   }
}
```

Now that you have the Kafka packages and configuration added to the project, you need to load the configuration details from C#.

__5. In VSCode, open kafka-worker-service\Program.cs

__6. Below the existing using statements, add the following namespace:

```
using Confluent.Kafka;
```

__7. In Program.cs, locate this line (around line 12):

```
builder.Services.AddSwaggerGen();
```

__8. Add these lines immediately after.

```
  var consumerConfig = new ConsumerConfig();

  builder.Configuration.Bind("ConsumerConfig", consumerConfig);

  builder.Services.AddSingleton<ConsumerConfig>(consumerConfig);
```

The above code will ensure that the appsettings.json configuration is loaded when the Web API is started/hosted.

__9. Right click **kafka-worker-service** and create a new folder named **Helpers**

__10. Using File Explorer, navigate to C:\LabFiles\kafka-worker-snippets and drag & drop ConsumerHelper.cs file to VSCode kafka-worker-service\Helpers folder.

__11. In VSCode, open **Helpers\ConsumerHelper.cs** and take a moment to review the code. (Note: The code is listed below for reference.):

```
 public class ConsumerHelper
    {
        private string _topicName;
        private IConsumer<string,string> _consumer;
        private ConsumerConfig _config;

        public ConsumerHelper(ConsumerConfig config,string topicName)
        {
            this._topicName = topicName;
            this._config = config;
            this._consumer = new ConsumerBuilder<string,
string>(this._config).Build();
            this._consumer.Subscribe(topicName);
        }
        public string ReadMessage()
        {
            var res = this._consumer.Consume();
            return res.Message.Value;
        }
    }
```

ConsumerHelper is a custom class that uses ConsumerConfig to read the configuration details that were read from appsettings.json and registered as a singleton object in the Startup.cs file. ConsumerBuilder uses the Kafka configuration to construct an IConsumer object and subscribes to a Kafka topic. The ReadMessage custom method uses the Consume method to read a message from the Kafka topic.

__12. In VSCode, open Services\UserRegistrationWorker.cs

__13. Below the existing using statements, add the following namespaces shown in **bold**:

```
using Microsoft.Extensions.Logging;
// TODO: add namespaces
using Newtonsoft.Json;
using Confluent.Kafka;
using kafka_worker_service.Helpers;
using kafka_worker_service.Models;
```

__14. Inside the UserRegistrationWorker class, add the code shown in **bold**:

```
    public class UserRegistrationWorker : BackgroundService
    {
        // TODO: appsettings dependency injection
        private readonly ConsumerConfig _config;

        public UserRegistrationWorker(ConsumerConfig config)
        {
              _config = config;
        }
```

The above lines ensure that the worker background service can read the Kafka cluster configuration from the appsettings.json file.

Part 5 - Add Logic to the Worker Project

In this part, you will write the logic to the Worker (BackgroundService) so it consumes messages from the user-registration Kafka topic.

__1. In VSCode, open kafka-worker-service\Services\UserRegistrationWorker.cs.

__2. Modify the ExecuteAsync method as shown in **bold**:

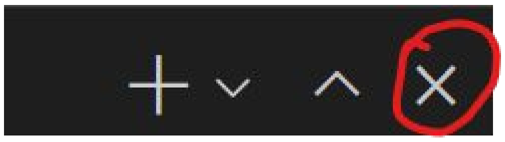
```
 protected override async Task ExecuteAsync(CancellationToken
stoppingToken)
{
    Console.WriteLine($"Worker started...");
    var consumerHelper = new ConsumerHelper(_config, "user-
registration");

    while (!stoppingToken.IsCancellationRequested)
    {
       // TODO: worker logic

       string message = consumerHelper.ReadMessage();

       //Deserilaize
       UserRegistrationModel user =
JsonConvert.DeserializeObject<UserRegistrationModel>(message);

       //TODO:: Process Order
       Console.WriteLine($"---Info: User Registered: {message}---");
    }
}
```

Note: The above lines use the custom helper class, ConsumerHelper, to read messages from the Kafka topic, deserialize them, and display them in the console.

Part 6 - Test the Consumer

In the previous labs, you have been using Confluent CLI to consume messages from the user-registration Kafka topic. Now that you have your custom consumer, you will test the consumer and see if it is able to consume messages from the Kafka topic.

__1. In VSCode, close all terminal windows by clicking Close.



__2. In VSCode, right click kafka-webapi-backend and select Open in Integrated Terminal.

__3. In the terminal window, execute the following command to start the Web API project:

```
dotnet watch run --urls=http://localhost:9090
```

__4. In VScode, right click kafka-aspnet-frontend and select Open in Integrated Terminal.

__5. In the terminal window, execute the following command to start the MVC project:

```
dotnet watch run --urls=http://localhost:8080
```

__6. In VScode, right click kafka-worker-service and select Open in Integrated Terminal.

__7. In the terminal window, execute the following command to start the Worker project. (Note: You don't need any specific port for the Worker since it will not be directly accessed by any client. Instead, it will continuously run in the background and consume messages from the Kafka topic):

```
dotnet watch run
```

__8. Switch to Chrome and navigate to the following URL:

```
http://localhost:8080
```

__9. Click the **Register** menu item.

__10. Enter values in to the fields and click **Register User**

__11. Switch to the terminal where the Worker project is running and ensure a message shows up in the terminal like this:

```
---Info: User Registered: {"UserId":"<Random GUID>","FirstName":"<Your
First name>","LastName":"<Your last name>","Email":"<Your email>"}---
```

__12. Also, check the terminal window where the kafka-webapi-backend is running and notice there is a message like this:

```
KAFKA => Delivered '{"UserId":"<Random GUID>","FirstName":"<Your first
name>","LastName":"<Your last name>","Email":"<Your email>"}' to 'user-
registration [[1]] @3'
```

TROUBLESHOOTING: If the worker project doesn’t consume and show messages right away, wait for up to a minute so it can catch up with the Kafka topic.

Part 7 - Clean-up

__1. Press **Ctrl+C** in all terminal windows to stop the Confluent CLI, kafka-webapi-backend, kafka-aspnet-frontend, and kafka-worker-service.

__2. Keep **VSCode** open for the next lab.

Part 8 - Review

In this lab, you created a .NET Core worker project that act as a Kafka client and consumed messages from the user-registration Kafka topic.

Lab 8 - Integrating Postgres SQL on AWS and Kafka

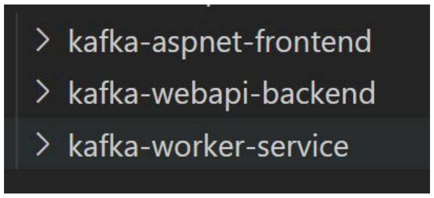
In this lab, you will continue with the previous lab and add Postgres SQL on AWS support to the existing to the worker project you developed in the previous lab.

Part 1 - Setup

__1. Using File Explorer, navigate to **C:\LabFiles** and extract the contents of **kafka-postgres-starter.zip** to **C:\LabWorks\kafka-postgres-starter**

__2. Open VSCode and from the menu click File → Open Folder and select C:\LabWorks\kafka-**postgres**-starter

__3. Ensure it has the following three projects:



```
> kafka-aspnet-frontend
> kafka-webapi-backend
> kafka-worker-service
```

Part 2 - A Table in Postgres SQL Database has been provided for you

The userregistration01 table has been created for you in a Postgres SQL database on AWS.

Part 3 - Configure the Worker Project to Support Postgres SQL via Entity Framework

In this part, you will configure the worker project you developed in the previous lab to support Postgre SQL via Entity Framework.

__1. In VSCode, right click kafka-worker-service and click Open in Integrated Terminal.

__2. In the terminal window, execute the following commands to add Entity Framework packages to the project:

```
dotnet add package Microsoft.EntityFrameworkCore.SqlServer --version
6.0.20

dotnet add package Microsoft.EntityFrameworkCore.Design  --version
6.0.20
```

```
dotnet tool install --global dotnet-ef --version 6.0.20

dotnet add package Npgsql.EntityFrameworkCore.PostgreSQL --version
7.0.11
```

__3. Execute the following command to ensure *dotnet-ef* is successfully installed:

```
dotnet-ef --version
```

__4. Ensure the following output is displayed:

```
Entity Framework Core .NET Command-line Tools
6.0.20
```

__5. Scaffolding for the entity framework is provided in the starter project Models directory. Expand the contents of the Models folder and explore UserRegistration01.cs.

__6. Execute the following command and ensure there are no errors:

```
dotnet build
```

By the end of this part, you have a .NET worker project with Entity Framework configured to connect to the Postgres SQL database.

Part 4 - Change the Worker Logic

In this part, you will modify the worker background service logic so it consumes messages from the user-registration Kafka topic and inserts them in Postgre SQL database table.

__1. In VSCode, review kafka-worker-service\Services\UserRegistrationWorker.cs.

__2. Open kafka-webapi-backend\appsettings.json

__3. Update the placeholders with values you used in the previous labs:

```
  "ConsumerConfig": {
    "BootstrapServers": "<Your Kafka Endpoint>",
    "SaslMechanism": "Plain",
    "SaslUsername": "<Your API Key>",
    "SaslPassword": "<Your API Secret>",
    "SecurityProtocol": "SaslSsl",
    "SslCaLocation": "probe",
    "GroupId": "kafka-lab"
```

```
}
```

__4. Open kafka-worker-service\appsettings.json

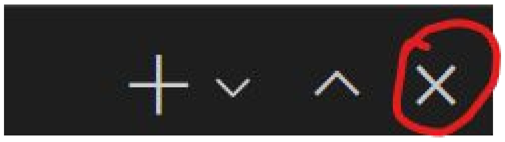
__5. Update the placeholders with values you used in the previous labs:

```
  },
  "AllowedHosts": "*",
  "ProducerConfig": {
    "BootstrapServers": "<Your Kafka Endpoint>",
    "SaslMechanism": "Plain",
    "SaslUsername": "<Your API Key>",
    "SaslPassword": "<Your API Secret>",
    "SecurityProtocol": "SaslSsl",
    "SslCaLocation": "probe"
    }
```

Part 5 - Test the Consumer

In this part, you will test the consumer worker project consumes messages from the Kafka topic and writes them to Postgre SQL database.

__1. In VSCode, close all terminal windows by clicking Close.



__2. In VScode, right click kafka-webapi-backend and click Open in Integrated Terminal.

__3. In the terminal window, execute the following command to start the Web API project:

```
dotnet watch run --urls=http://localhost:9090
```

__4. In VScode, right click kafka-aspnet-frontend and click Open in Integrated Terminal.

__5. In the terminal window, execute the following command to start the MVC project:

```
dotnet watch run --urls=http://localhost:8080
```

__6. In VScode, right click kafka-worker-service and click Open in Integrated Terminal.

__7. In the terminal window, execute the following command to start the Worker project. (Note: You don't need any specific port for the Worker since it will not be directly accessed by any client. Instead, it will continuously run in the background and consume messages from the Kafka topic):

```
dotnet watch run
```

__8. Switch to Chrome and navigate to the following URL:

```
http://localhost:8080
```

__9. Click the **Register** menu item.

__10. Enter values into the fields and click **Register User**.

__11. Switch to the terminal where the Worker project is running and ensure a message shows up in the terminal like this:

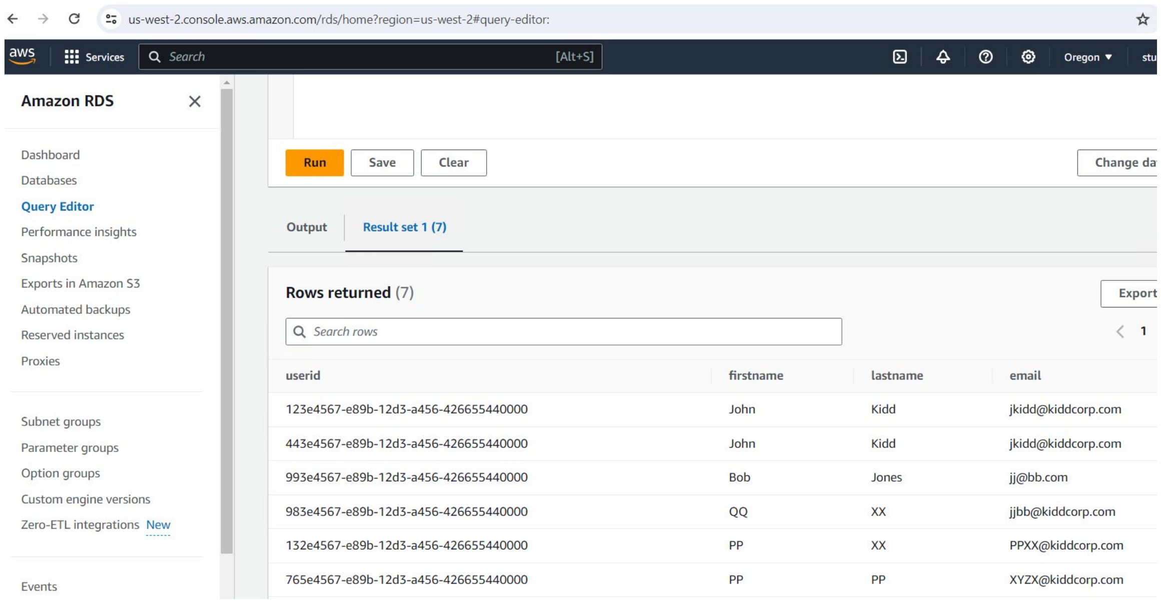
```
---Info: User Registered: {"UserId":"<Random GUID>","FirstName":"<Your 
First name>","LastName":"<Your last name>","Email":"<Your email>"}---
```

__12. Also, check the terminal window where the kafka-webapi-backend is running and notice there is a message like this:

```
KAFKA => Delivered '{"UserId":"<Random GUID>","FirstName":"<Your first 
name>","LastName":"<Your last name>","Email":"<Your email>"}' to 'user-
registration [[1]] @3'
```

TROUBLESHOOTING: If the worker project doesn't consume and show messages right away, wait for up to a minute so it can catch up with the Kafka topic.

Your instructor can walk you through the steps to verify you have a record in the Postgres SQL Database on AWS.



Rows returned (7)

userid	firstname	lastname	email
123e4567-e89b-12d3-a456-426655440000	John	Kidd	jkidd@kiddcorp.com
443e4567-e89b-12d3-a456-426655440000	John	Kidd	jkidd@kiddcorp.com
993e4567-e89b-12d3-a456-426655440000	Bob	Jones	jj@bb.com
983e4567-e89b-12d3-a456-426655440000	QQ	XX	jjbb@kiddcorp.com
132e4567-e89b-12d3-a456-426655440000	PP	XX	PPXX@kiddcorp.com
765e4567-e89b-12d3-a456-426655440000	PP	PP	XYZX@kiddcorp.com

Part 6 - Clean-up

__1. Press **Ctrl+C** in all terminal windows to stop the Confluent CLI, kafka-webapi-backend, kafka-aspnet-frontend, and kafka-worker-service.

__2. Keep **VSCode** open for the next lab.

Part 7 - Review

In this lab, you added support for Azure SQL to the worker project you created in the previous lab.

Lab 9 - Confluent Kafka Connect

In this Lab you will leave how to work with Confluent Kafka Connect to transfer data.

Part 1 - Using Confluent Kafka Connect

Writing data with a C# Producer and Consumer is a convenient place to start, but you will probably want to use data from other sources or export data from Kafka to other systems. For many systems, instead of writing custom integration code you can use Kafka Connect to import or export data.

Kafka Connect is a tool included with Kafka that imports and exports data to Kafka. It is an extensible tool that runs *connectors*, which implement the custom logic for interacting with an external system. Kafka Connect is designed to be extensible so that developers can create custom connectors, transforms, and converters, and users can install and run them.

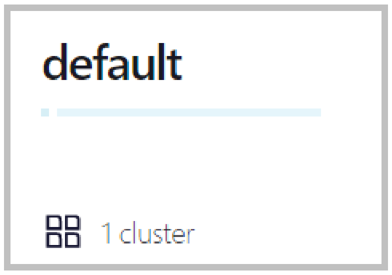
In this part, you will see how to run Kafka Connect with simple connectors that import data from a datagen file to a Kafka topic and export data from a Kafka topic to a file.

__1. Open your browser and login to the Confluent Cloud site with your credential, if not already there.

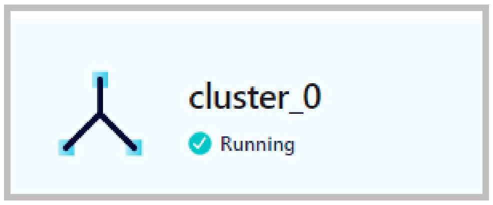
`https://confluent.cloud/`

__2. On the left-hand side, click **Environments**.

__3. Click on your default cluster, your unique named cluster.

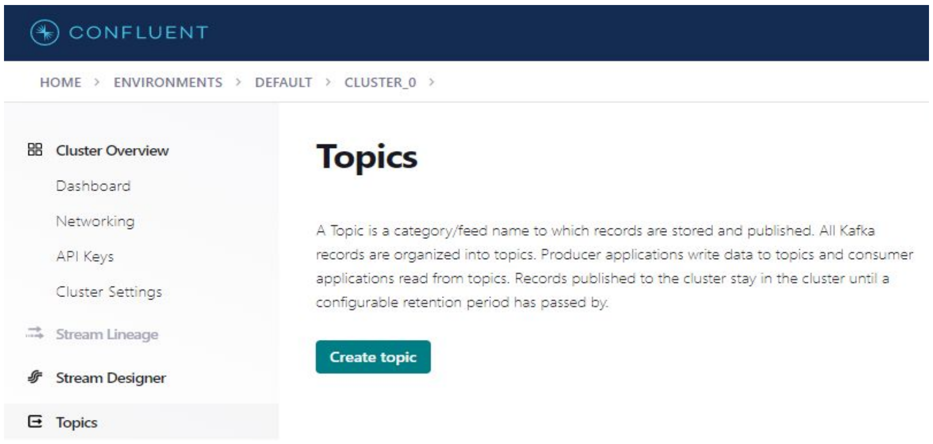


__4. Click on your cluster.



__5. On the left-hand menu select the **Topics** link.

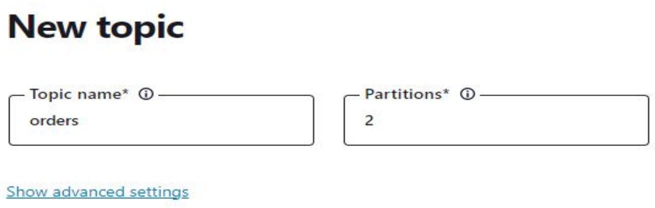
__6. If you don't have any Topic, click **Create topic** as shown below, if you already have topics then click **Add topic**.



Topics

A Topic is a category/feed name to which records are stored and published. All Kafka records are organized into topics. Producer applications write data to topics and consumer applications read from topics. Records published to the cluster stay in the cluster until a configurable retention period has passed by.

__7. Name the topic ***orders*** and ensure that the Number of partitions is set to **2**.



New topic

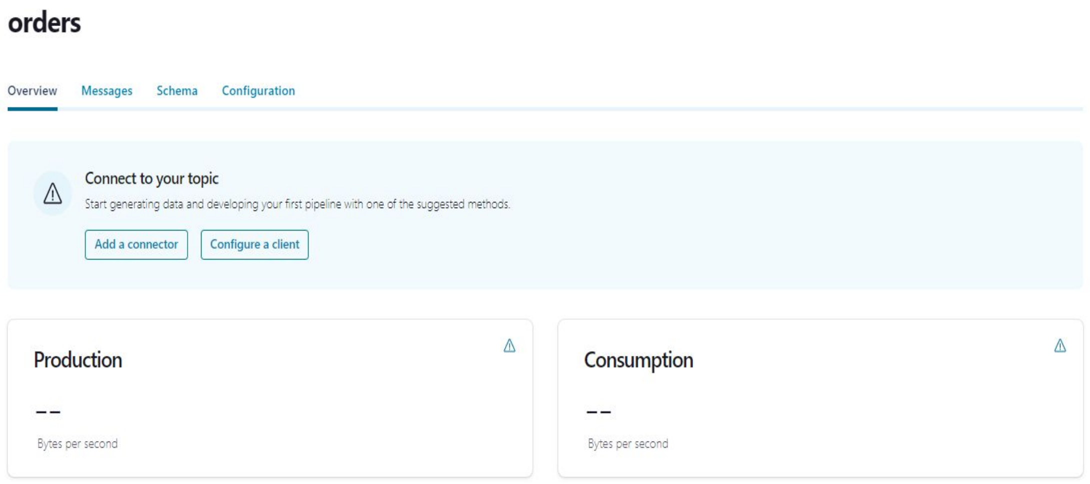
Topic name*: orders

Partitions*: 2

__8. Review the default advanced settings information by clicking the **Show advanced settings** button.

__9. When done, click **Save & create**.

The resulting screen shows the current state of your new topic. Use the tabs to familiarize yourself with the available options here like Messages, Schema and Configuration. Once we have set up the Connector we will review these further. As you can see, nothing is currently being produced or consumed for the topic orders.


orders

Overview Messages Schema Configuration

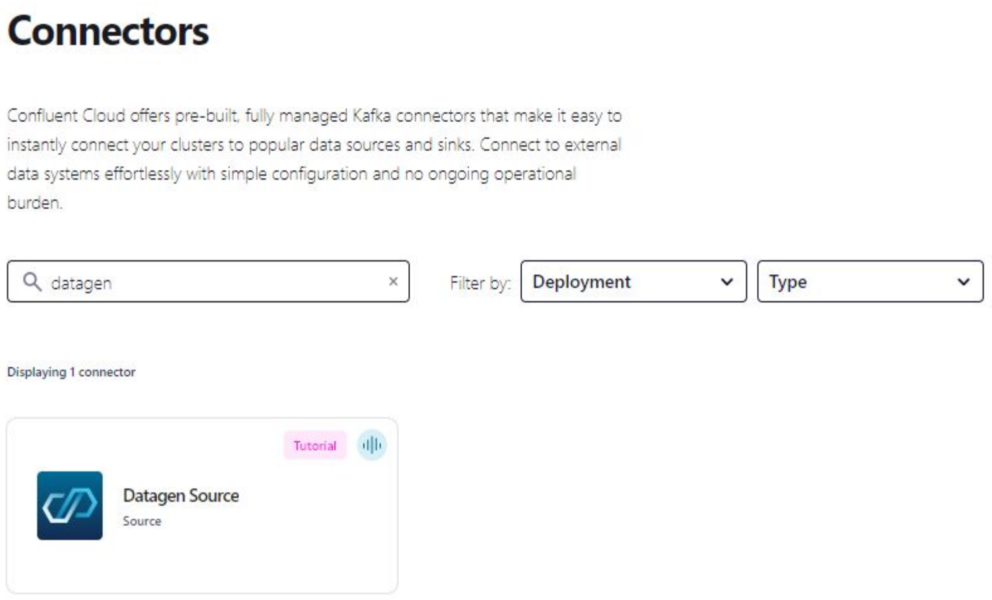
Connect to your topic

Start generating data and developing your first pipeline with one of the suggested methods.

__10. Click the **Add a connector** button or from the left-hand menu select **Connectors**.

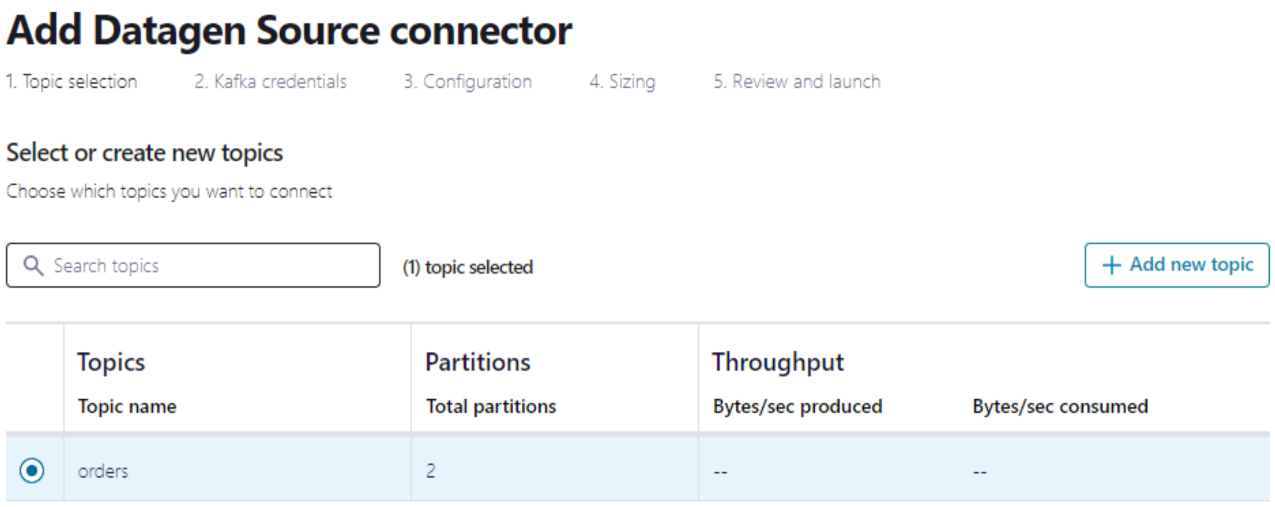
Confluent offers over 120 connectors to choose from, or you can customize this by creating your own. We will work with the **Datagen Source** connector in this Lab.

__11. In the search selector type **datagen** to find it easily. Then click on the resulting icon.


Connectors

Confluent Cloud offers pre-built, fully managed Kafka connectors that make it easy to instantly connect your clusters to popular data sources and sinks. Connect to external data systems effortlessly with simple configuration and no ongoing operational burden.

__12. In the resulting screen select the **orders** topic you created previously. Now click **Continue**.



Add Datagen Source connector

1. Topic selection 2. Kafka credentials 3. Configuration 4. Sizing 5. Review and launch

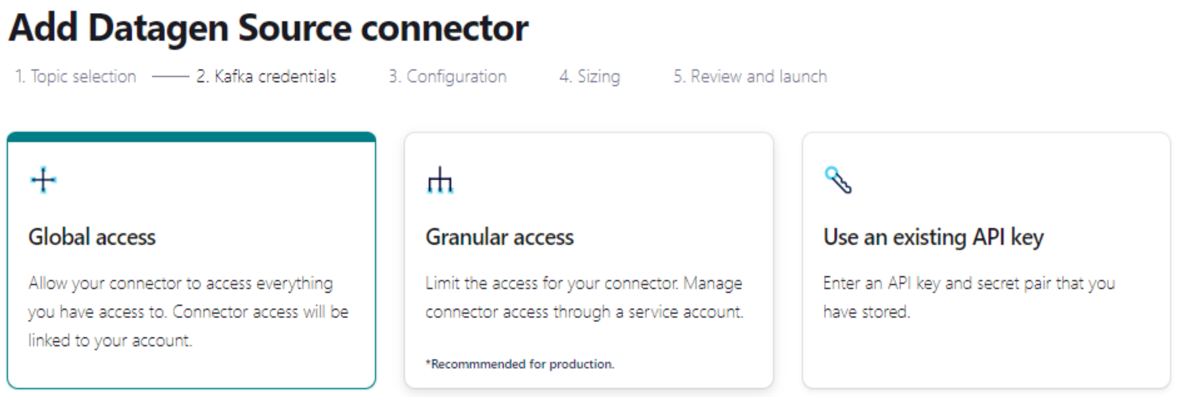
Select or create new topics

Choose which topics you want to connect

Search topics (1) topic selected + Add new topic

	Topics	Partitions	Throughput	
	Topic name	Total partitions	Bytes/sec produced	Bytes/sec consumed
●	orders	2	--	--

Confluent Kafka offers granular access to topics for both producer and consumer connections. Here we are presented with three options for setting up access to the topic orders. Your organization would likely enforce a policy that may vary depending upon the security required of the data being transferred. Kafka does support TLS and encryption of topics as required. The default is *Global access* highlighted below.



Add Datagen Source connector

1. Topic selection 2. Kafka credentials 3. Configuration 4. Sizing 5. Review and launch

Global access

Allow your connector to access everything you have access to. Connector access will be linked to your account.

Granular access

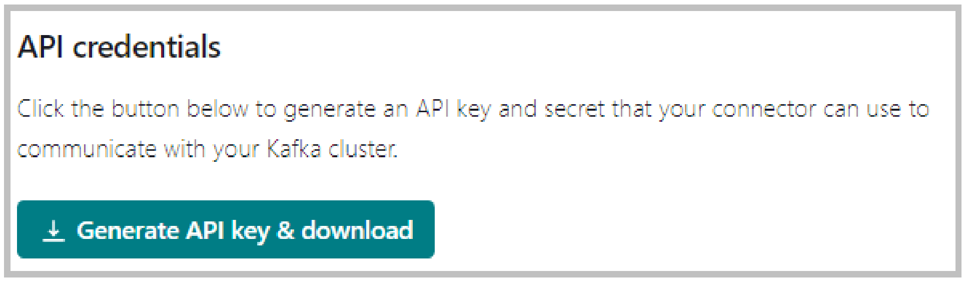
Limit the access for your connector. Manage connector access through a service account.

*Recommmended for production.

Use an existing API key

Enter an API key and secret pair that you have stored.

__13. Scroll down and click Generate API key & download.

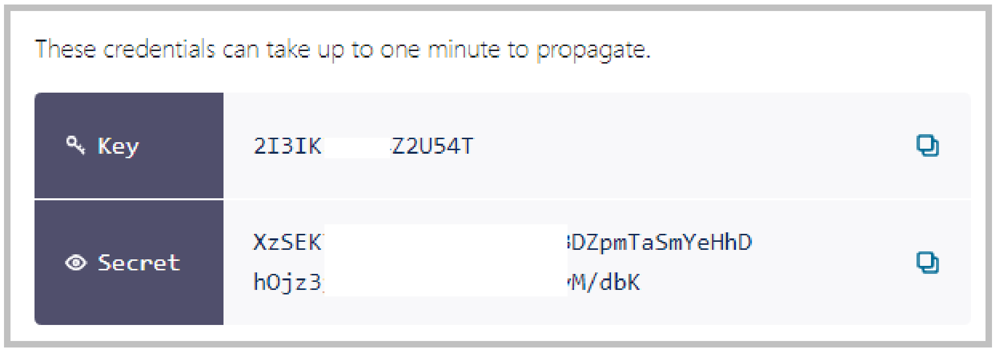


API credentials

Click the button below to generate an API key and secret that your connector can use to communicate with your Kafka cluster.

Generate API key & download

Credentials will be shown, and a file will be downloaded.



These credentials can take up to one minute to propagate.

Key	2I3IK [redacted] Z2U54T
Secret	XzSEK [redacted] DZpmTaSmYeHhD h0jz3 [redacted] M/dbK

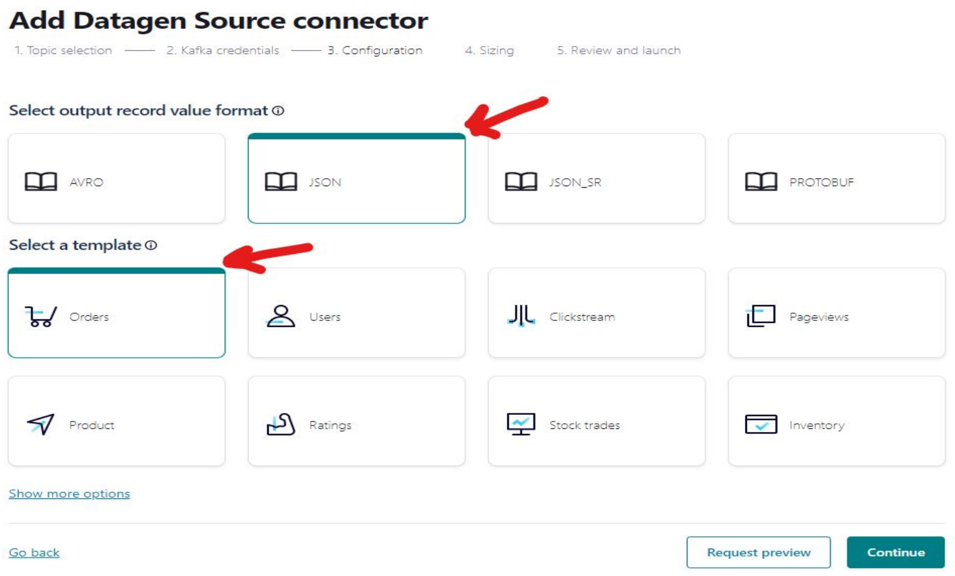
__14. Click Continue

Confluent delivers many options for the user in setting up connectors. In fact, there are over 40 choices within this section available for configuration. We will pick two items that will make our viewing the output later easier, you can always return later and edit the choices to see the changed output.

__15. For the output record format select **JSON**

__16. For the template utilize **Orders** since that is the data we are generating in this example.

You may wish to review the other options, expand this using the Show more options and the Advanced configurations setting. In the latter section you can set the decimal format note it is BASE64 by default but can be updated to Numeric. You can also set a Schema Keyfield, optional here but generally best practice. Finally setting of the interval between messages, currently 1000 ms and Transforms can be applied for conditional matching of messages.



Add Datagen Source connector

1. Topic selection — 2. Kafka credentials — 3. Configuration — 4. Sizing — 5. Review and launch

Select output record value format

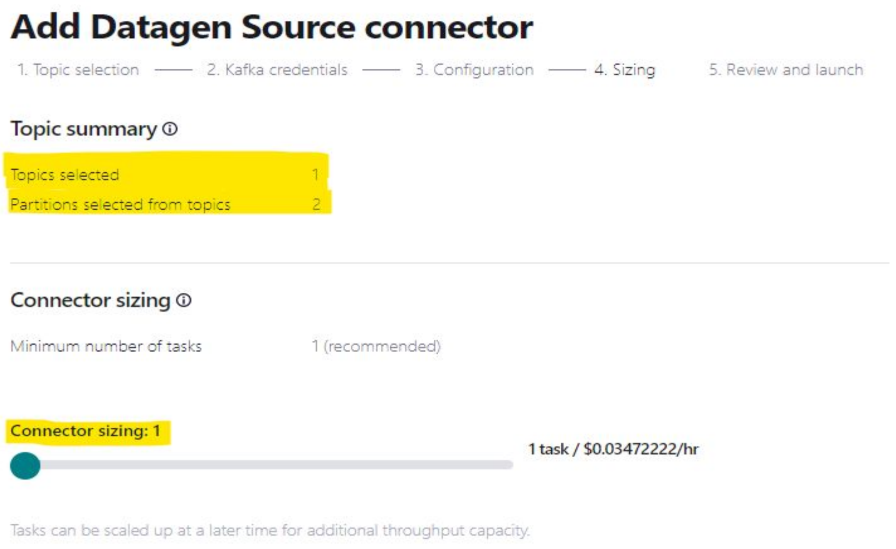
AVRO | [✓ JSON] | JSON_SR | PROTOBUF

Select a template

[✓ Orders] | Users | Clickstream | Pageviews | Product | Ratings | Stock trades | Inventory

__17. Click Continue.

The resulting page shows the current configuration involving the messaging and the Connector sizing. There should be 1 topic with two partitions and the connector sizing should be 1 in this example.



Add Datagen Source connector

1. Topic selection —— 2. Kafka credentials —— 3. Configuration —— 4. Sizing   5. Review and launch

Topic summary ⓘ

Topics selected	1
Partitions selected from topics	2

Connector sizing ⓘ

Minimum number of tasks	1 (recommended)

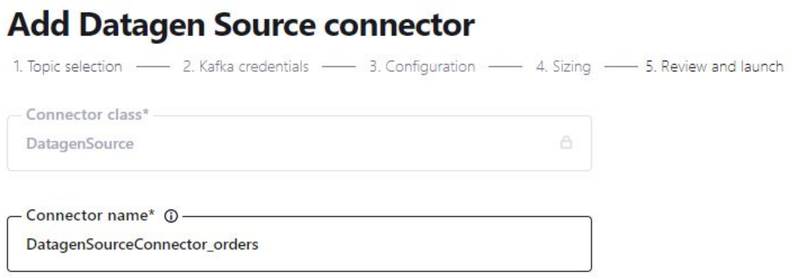
Connector sizing: 1

1 task / $0.03472222/hr

Tasks can be scaled up at a later time for additional throughput capacity.

__18. Click Continue.

__19. In the window we can review the current configuration items. Update the DatagenSourceConnector_0 name to **DatagenSourceConnector_orders**



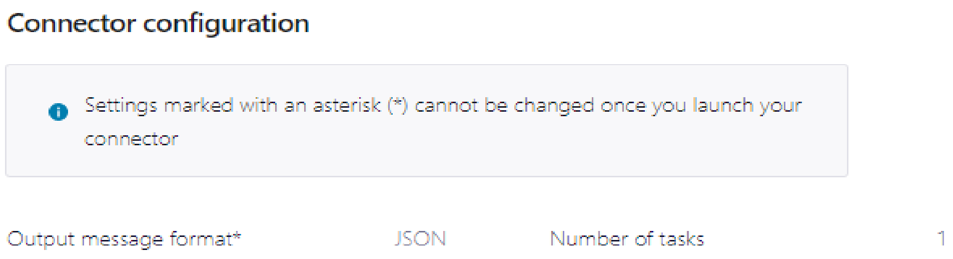
Add Datagen Source connector

1. Topic selection —— 2. Kafka credentials —— 3. Configuration —— 4. Sizing —— 5. Review and launch

Connector class*: DatagenSource

Connector name* ⓘ: DatagenSourceConnector_orders

In the Configurations & cost section, note the output message format is JSON.

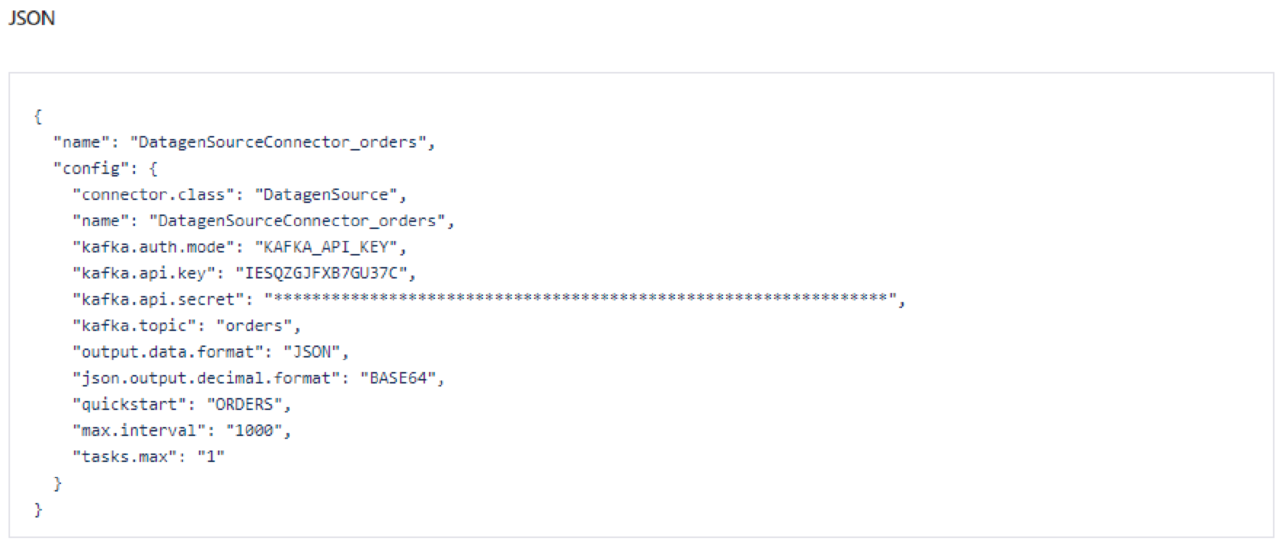


Connector configuration

Settings marked with an asterisk (*) cannot be changed once you launch your connector

Output message format*	JSON	Number of tasks	1

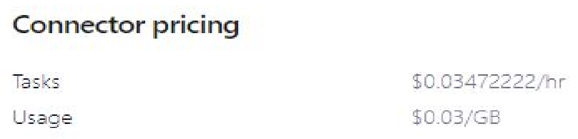
The actual output file configuration is shown below this in JSON format. Note the key and secret we automatically generated for us, we could have downloaded this information earlier but don't need it for this example as everything is running within the Confluent Cloud.



JSON

```
{
  "name": "DatagenSourceConnector_orders",
  "config": {
    "connector.class": "DatagenSource",
    "name": "DatagenSourceConnector_orders",
    "kafka.auth.mode": "KAFKA_API_KEY",
    "kafka.api.key": "IESQZGJFXB7GU37C",
    "kafka.api.secret": "****************************************************************",
    "kafka.topic": "orders",
    "output.data.format": "JSON",
    "json.output.decimal.format": "BASE64",
    "quickstart": "ORDERS",
    "max.interval": "1000",
    "tasks.max": "1"
  }
}
```

Notice that every connector you generate will provide a cost estimate for usage. Actual costing will vary with the amount of data in use, transformations performed and of course throughput.



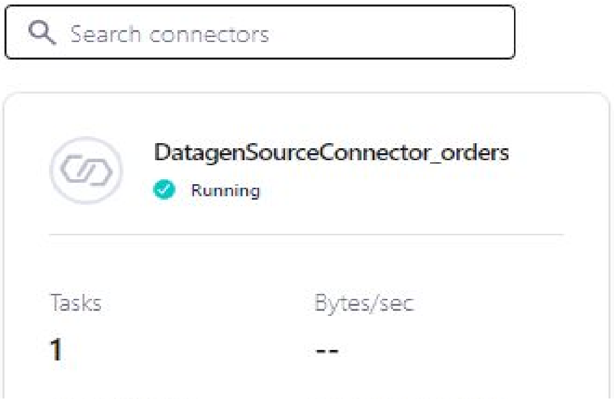
Connector pricing

Tasks	$0.03472222/hr
Usage	$0.03/GB

__20. Click **Continue** when ready.

__21. Click **Allow notifications** in the **Enable browser notifications** dialog.

__22. You should be taken to the Connectors window and the newly created DatagenSourceConnector_orders should be there and running. If this is not the case, let your instructor know.



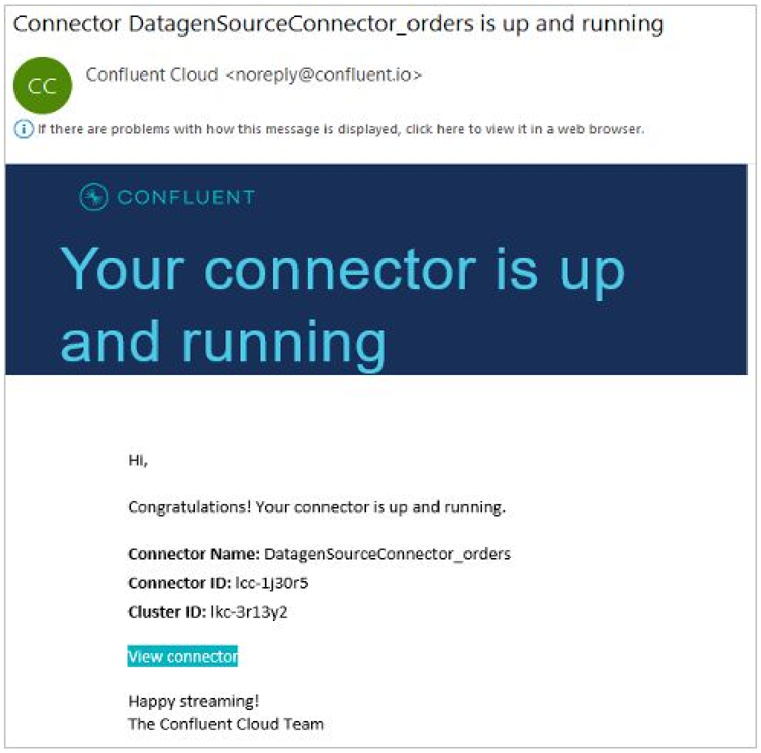
DatagenSourceConnector_orders

Running

Tasks: 1

Bytes/sec: --

Additionally, you probably received an email from Confluent Cloud stating this Connector is available similar to below.



Connector DatagenSourceConnector_orders is up and running

Confluent Cloud <noreply@confluent.io>

If there are problems with how this message is displayed, click here to view it in a web browser.

CONFLUENT

Your connector is up and running

Hi,

Congratulations! Your connector is up and running.

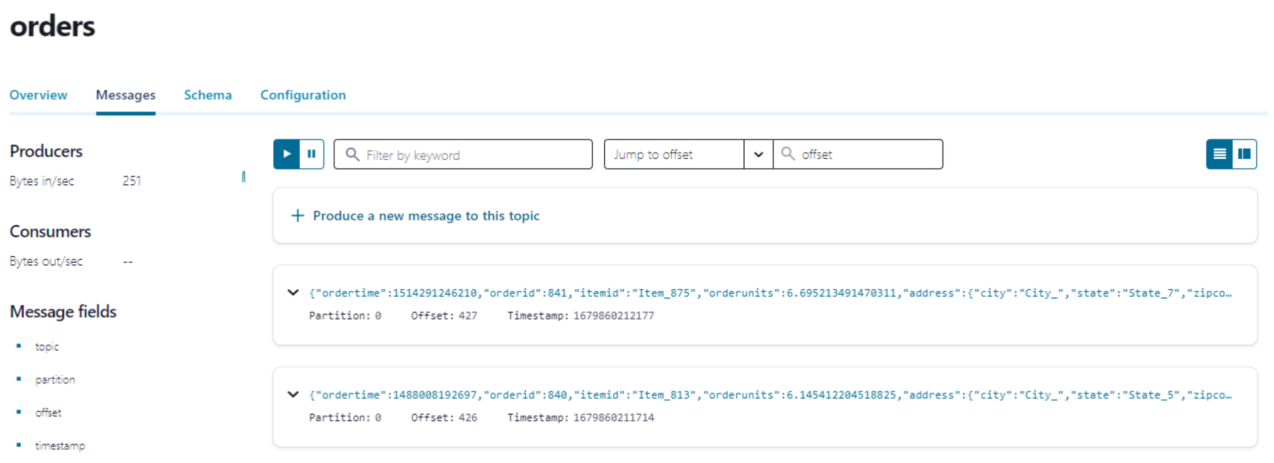
Connector Name: DatagenSourceConnector_orders
Connector ID: lcc-1j30r5
Cluster ID: lkc-3r13y2

View connector

Happy streaming!
The Confluent Cloud Team

__23. From the **Topics** page of your cluster, select the **orders** topic and then the **Messages** tab.

You should see a stream of new messages arriving every few seconds. Explore the output as desired, review the Messages, Schema and Configurations tabs.



```
{"ordertime":1514291246210,"orderid":841,"itemid":"Item_875","orderunits":6.695213491470311,"address":{"city":"City_","state":"State_7","zipco…
Partition: 0     Offset: 427     Timestamp: 1679860212177

{"ordertime":1488008192697,"orderid":840,"itemid":"Item_813","orderunits":6.145412204518825,"address":{"city":"City_","state":"State_5","zipco…
Partition: 0     Offset: 426     Timestamp: 1679860211714
```

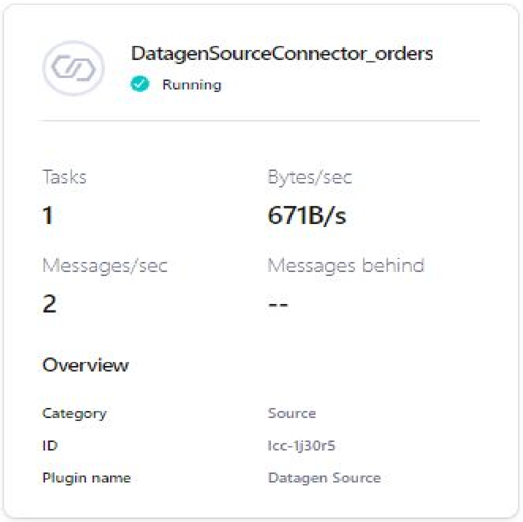
Congratulations you have successfully used Confluent Kafka Connector DataGen to produce topic messages for your orders topic.

Part 2 - Shutdown the Confluent Kafka Connect

__1. Return to the **Connectors** window via the left-hand menu choice.

In the Connectors page, you should see updated numbers on how many bytes per second are being transferred like below.

__2. Click on your connector we created earlier called **DatagenSourceConnector_orders**.

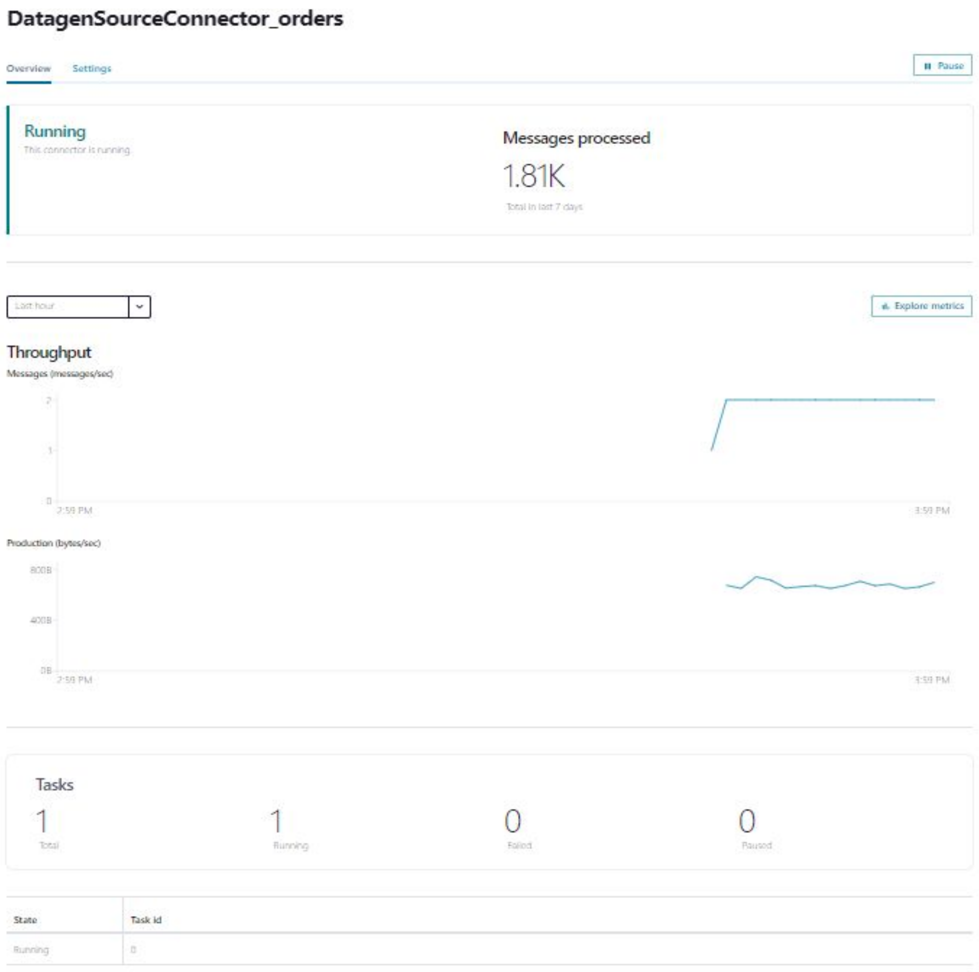


Tasks	Bytes/sec
1	671B/s
Messages/sec	Messages behind
2	--

Overview

Category	Source
ID	lcc-1j30r5
Plugin name	Datagen Source

__3. In this window there are two tabs to choose from: Overview and Settings. Review the tabs. Specifically look at the throughput graph and number of running tasks.



DatagenSourceConnector_orders

Running
This connector is running

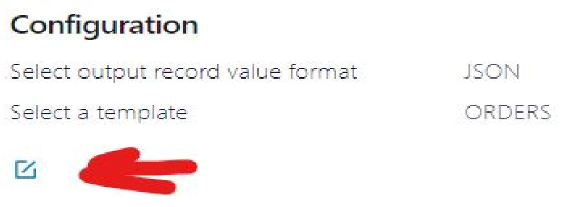
Messages processed
1.81K
Total in last 7 days

Tasks

1	1	0	0
Total	Running	Failed	Paused

State	Task id
Running	0

__4. In the setting window, you can see the configuration choices we made earlier, and these may be modified using the **Edit** buttons. You could change the output type to **AVRO** and review the topics again via the Topic menu if desired. Don't forget to click **Apply changes** first.



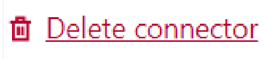
Configuration

Select output record value format	JSON
Select a template	ORDERS

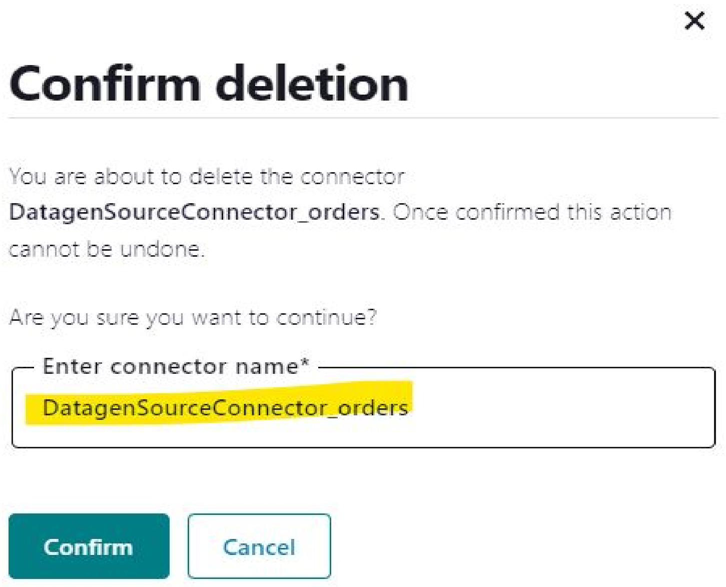
From the Connector | Settings window you also have the ability to pause a connector using the **Pause** button on the top right side. The connector will go from Running to Paused state.

Clicking the **Resume** button in the Connector | Settings window in the previous location of the Pause button would restart the connector functionality and messages flowing.

__5. Finally, we will clean up our DatagenSourceConnector_orders connector and remove the Topic orders. On the Connector | Settings window at the bottom left side click **Delete Connector**.



__6. A popup will ask to confirm the deletion by adding the connector's name. Click **Confirm**.


Confirm deletion

You are about to delete the connector **DatagenSourceConnector_orders**. Once confirmed this action cannot be undone.

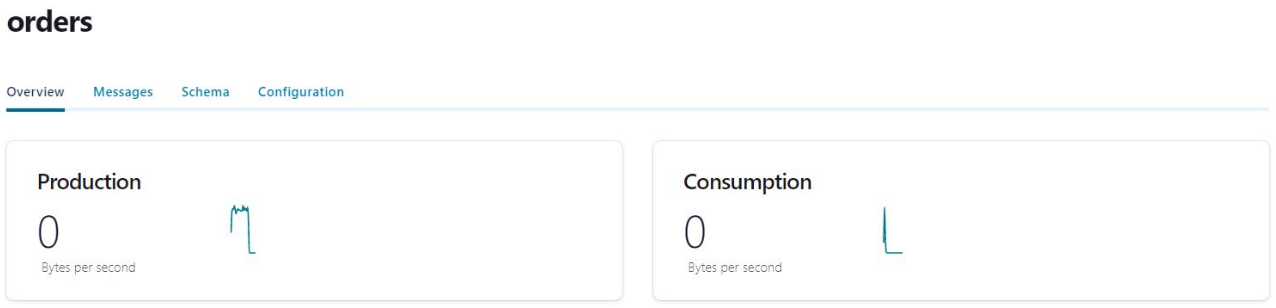
Are you sure you want to continue?

Enter connector name*: DatagenSourceConnector_orders

You will be returned to the Connector window and no running connectors will be shown.

__7. Click **Topics** and open the **orders** topic by clicking on it.

You should see something similar to below for your output between Production and Consumption of messages. Again, review the Messages, Schema and Configurations tabs as before.



__8. On the **Configuration** tab, choose the **Delete** topic button to remove the topic.

__9. Complete the information in the popup with topic name **orders** and then click **Continue**.

Part 3 - Review

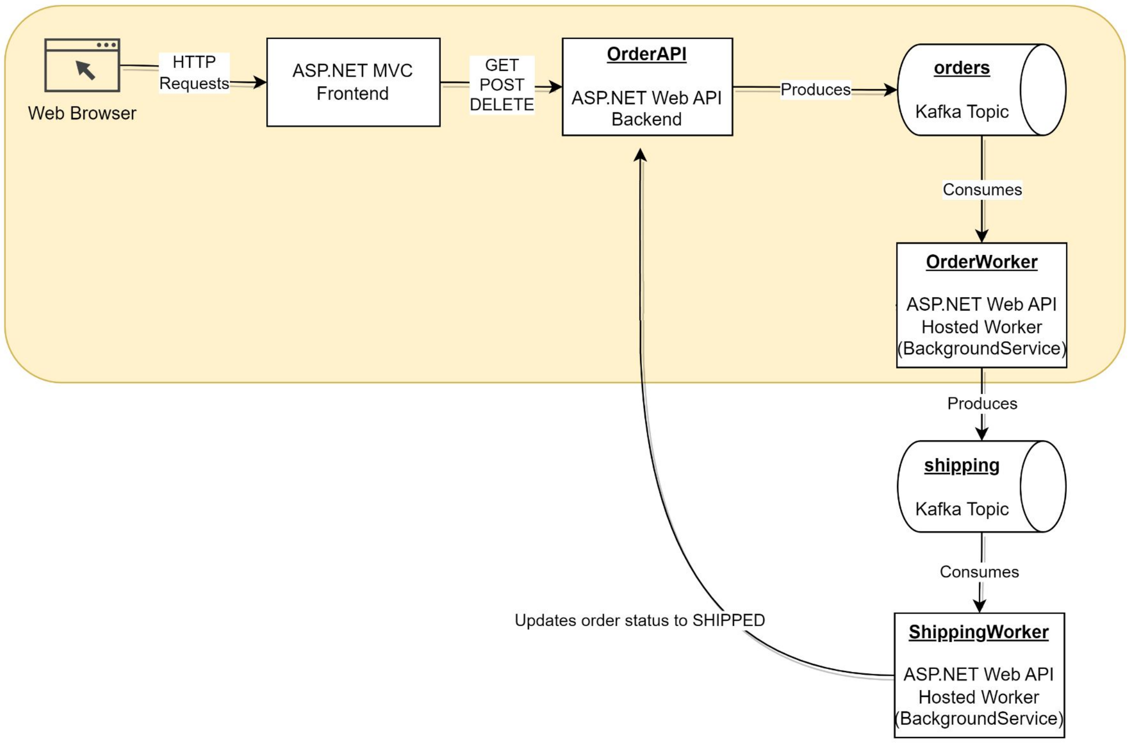
In this section, you learned the methodology to work with Kafka Connect. We created topic data in the DataGen Tool and retrieved JSON messages via a Kafka Connector.

Lab 10 - The Final Project

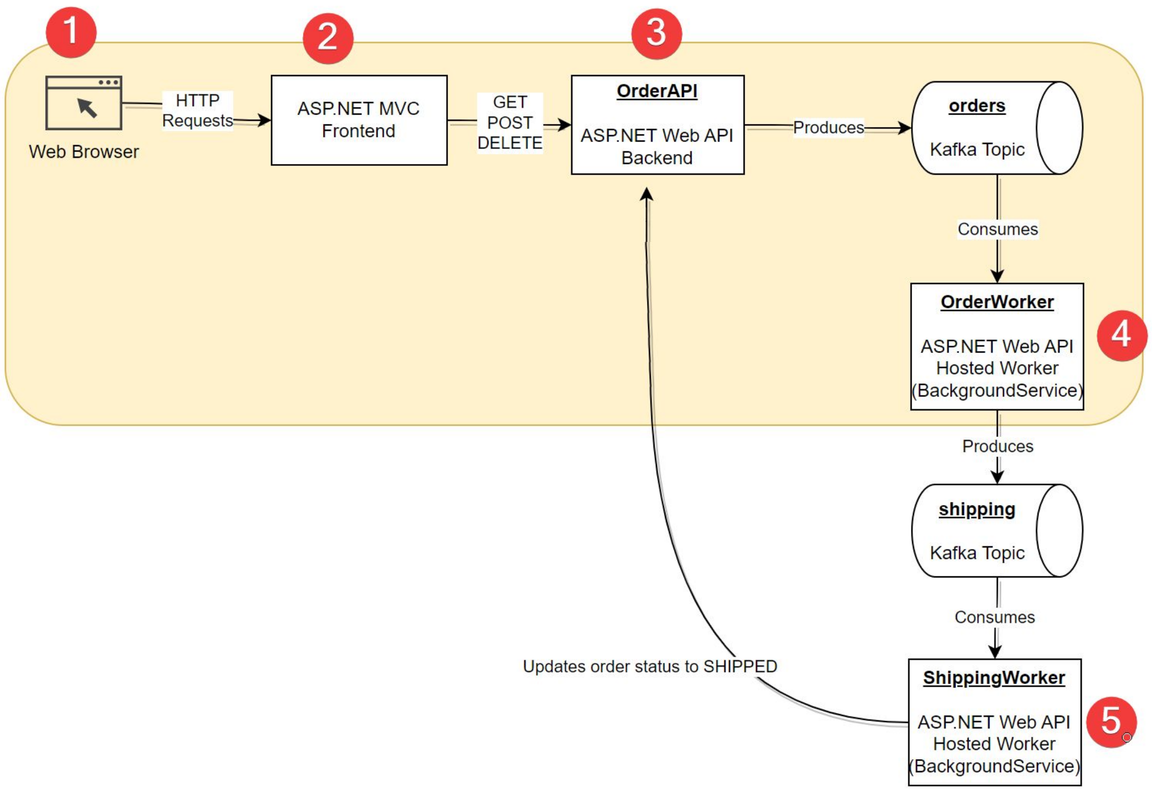
Now that we've seen how to combine ASP.NET Core Web API, ASP.NET Core MVC, and Kafka, it's your turn to finish the Final Project by creating the services to meet the Kafka Microservice requirements in the following sections.

Part 1 - Overall Architecture

Note: The *mandatory* part of the project is highlighted in yellow. Whatever is outside of the yellow block is *optional*.



Order Placement - Workflow Diagram and Explanation



__1. A user places an order request from a web browser.

__2. The ASP.NET MVC front-end application receives the order request and calls the OrderAPI with the order information captured from the end-user.

__3. OrderAPI receives the POST request from the MVC frontend and writes the order details to the orders Kafka topic. The initial order status is set to PENDING.

__4. OrderWorker is a Kafka consumer. The OrderWorker worker/background service continuously runs and consumes messages/orders from the orders Kafka topic. The order status is initially set to PLACED.

__5. (**OPTIONAL**): OrderWorker can also act as a Kafka Producer and write order details to the shipping Kafka topic.

__6. (**OPTIONAL**): ShippingWorker is a Kafka consumer. The ShippingWorker worker/background service continuously runs and consumes messages/orders from the shipping Kafka topic. The worker updates the order status from PLACED to SHIPPED.

Part 2 - Order Deletion - Workflow Diagram and Explanation

__1. The end-user views the orders list page that shows all orders with the Delete button next to each order. The user clicks the Delete button to delete an order.

__2. The MVC frontend receives the Order Id of the record to be deleted and calls the OrderAPI Web API.

OrderAPI queries checks the following:

__3. If the status is PENDING, OrderAPI writes the order details to the orders Kafka topic.

__4. If the status is PLACED or SHIPPED, OrderAPI responds to the client with the message "The order has already been <Order Status | PLACED or SHIPPED> and cannot be deleted."

__5. OrderWorker consumes messages/orders from the orders Kafka topic. The worker deletes the order from the table. Your challenge here is to identify how to differentiate between the INSERT (POST) and DELETE operations. Hint! See if you can pass both the order details and the operation you want to perform on the order.

Part 3 - Orders View - Workflow Diagram and Explanation

__1. The end-user views the Order List page.

__2. The MVC frontend sends the request to OrderAPI.

__3. OrderAPI gets the order list from Azure SQL using Entity Framework and returns it to the MVC frontend and the MVC frontend displays it on the View page.

Part 4 - Starter Project

Complete your project under the **C:\LabWorks\Project** folder. You have been provided with the following starter projects in the **C:\LabFiles\Project** folder. (Note: Copy them to C:\LabWorks\Project folder.

- kafka-project-mvc (ASP.NET MVC frontend)
  - The controller (.cs) and views (.cshtml) are implemented but most of the code is commented out. Don't forget to uncomment the code as you see fit after completing the backend.
- kafka-project-order-webapi (ASP.NET Web API)
  - The OrderAPI already has the GET, POST, and DELETE methods with the request/response handling. Most of the code is commented out. Don't forget to uncomment the code as you see fit when you implement these methods.

NOTE: Attempt the project on your own first. The sample solution for the project is available in the **C:\LabFiles\Solution** folder. Kafka consumer and producer configuration is removed from the appsettings.json file. Since this is a Kafka course, we are only covering the Kafka best practices, e.g. configuring the cluster connectivity information in appsettings.json.

kafka-project-mvc (ASP.NET MVC frontend) - GUI

The frontend has two important views:

Place Order

This view allows a user to create a new order. Only Customer Name, Order Description, and Order Amount can be configured by the user.

View Orders

This view allows the user to view records with details, such as Order Id, Order Date, and Order Status, that are generated by the backend and cannot be modified by the user. and delete a record.

The user can also send the order deletion request from here.

kafka-project-order-webapi (ASP.NET Web API)

The OrderAPI controller is available in this project. It contains the following endpoints and operations:

Function Name	Endpoint	Method	Description
GetOrders	/api/order	GET	In the starter project, it returns a list of hard-coded records.
GetOrderById	/api/order/{id}	GET	In the starter project, it returns a hard-coded record when an Order Id is passed to it.
PostOrder	/api/order	POST	In the starter project, it receives Order details from the request body and adds it to a hard-coded in-memory list.
DeleteOrderById	/api/order/{id}	DELETE	In the starter project, it receives the Order id and deletes it from the in-memory list.

Part 5 - Project Requirements

Here are the project requirements:

__1. Order topic

a. Create a topic named Order to store information from Kafka. The structure would be:

OrderId

CustomerName

OrderDescription

OrderDate

OrderAmount

OrderStatus

Status can be one of the following: (**Note**: You don't need to create a constraint on the column in the database but ensure these are enforced in C#.)

PENDING

PLACED

SHIPPED

__2. ASP.NET MVC Frontend

a. The frontend calls the OrderAPI using a helper class with GET, POST, and DELETE methods.

__3. OrderAPI

a. Implement the GET methods for GetOrders and GetOrderById to retrieve data from the Order topic.

b. Implement the POST method so it writes order details to the orders Kafka topic with the default status of PENDING.

c. Implement the DELETE method. The method should use the Kafka Order topic and check the order.

i. If the Order has the status of SHIPPED, return the message "The order has already been SHIPPED and cannot be deleted."

ii. If the Order status is PENDING, it should write it to the order Kafka topic.

__4. **orders** - Create the orders Kafka topic.

__5. OrderWorker

a. Create a .NET Web API hosted worker named OrderWorker. It should be a background service that continuously runs and consumes messages from the orders Kafka topic.

b. OrderWorker should check if the order is supposed to be inserted or deleted.

c. OrderWorker should perform the order insertion or deletion operation.

d. (OPTIONAL): OrderWorker should also act as a Kafka producer and write the order details to the shipping Kafka topic.

__6. (OPTIONAL) **shipping** - create the shipping Kafka topic.

__7. (OPTIONAL) ShippingWorker

a. (OPTIONAL) Create a .NET Web API hosted worker named ShippingWorker. Ensure you host it in a separate .NET Web API project. It should be a background service that continuously runs and consumes messages from the orders Kafka topic.

b. (OPTIONAL) ShippingWorker should use Kafka to change the order status to SHIPPED.

Part 6 - Running the Project Components

To run the various components of the project, use the commands mentioned in the table below.

Projects Component	Command
kafka-project-mvc	dotnet watch run --urls=http://localhost:8080
kafka-project-order-webapi (OrderAPI)	dotnet watch run --urls=http://localhost:9090
kafka-project-order-worker (OrderWorker)	dotnet watch run --urls=http://localhost:9091
kafka-project-shipping-worker (ShippingWorker)	dotnet watch run --urls=http://localhost:9092

Part 7 - Tips

__1. To create a Web API project, use dotnet new webapi --output my-project-name.

__2. Creating Kafka topics using Confluent CLI is covered in the **Creating an ASP.NET Core MVC Kafka Client** lab. Creating Kafka topics using Confluent Cloud Console (GUI) is covered in the **Understanding Kafka Topics** lab.

__3. Kafka configuration details related to appsettings.json and Startup.cs and covered in various labs, such as **Creating a .NET Core 3.1 Worker Kafka Client** and **Creating a .NET Core 3.1 ASP.NET Web API Kafka Client.**

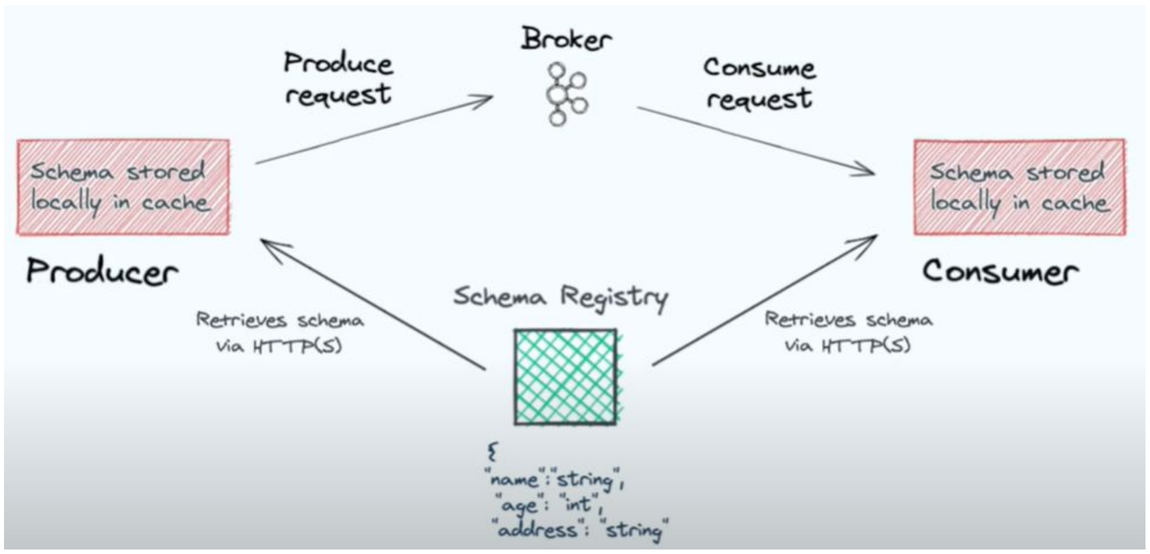
__4. NOTE: If you don’t see messages produced/consumes in Kafka topics and there is no issue with the code, wait for 3-4 minutes so the producer/consumer can sync with Kafka topics.

Lab 11 - BONUS - Kafka Schema Registry

In this Lab you will learn how to work with Confluent Kafka Schema Registry.

Part 1 - Using Confluent Kafka Schema Registry

The Schema Registry provides a location for managing and validating schemas for topic message data. Producers and consumers can use schemas associated directly with a topic for data consistency and compatibility as schemas evolve. This is a key component for data governance today. Schema Registry ensures data quality, supports standards and data lineage, along with audit capabilities, collaboration across teams, efficient application development protocols, and better overall system performance.


Broker
Produce request
Consume request
Schema stored locally in cache
Schema stored locally in cache
Producer
Consumer
Schema Registry
Retrieves schema via HTTP(S)
Retrieves schema via HTTP(S)
{
"name":"string",
"age": "int",
"address": "string"

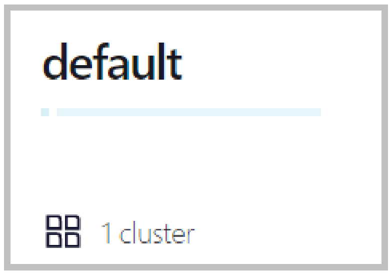
Source: Confluent

__1. Open your browser and login to the Confluent Cloud site with your credential, if not already there.

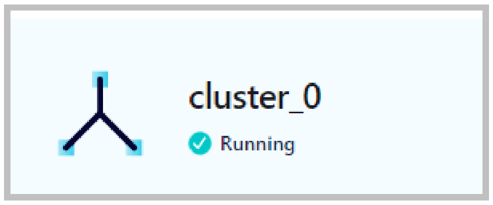
https://confluent.cloud/

__2. On the left-hand side, click **Environments**.

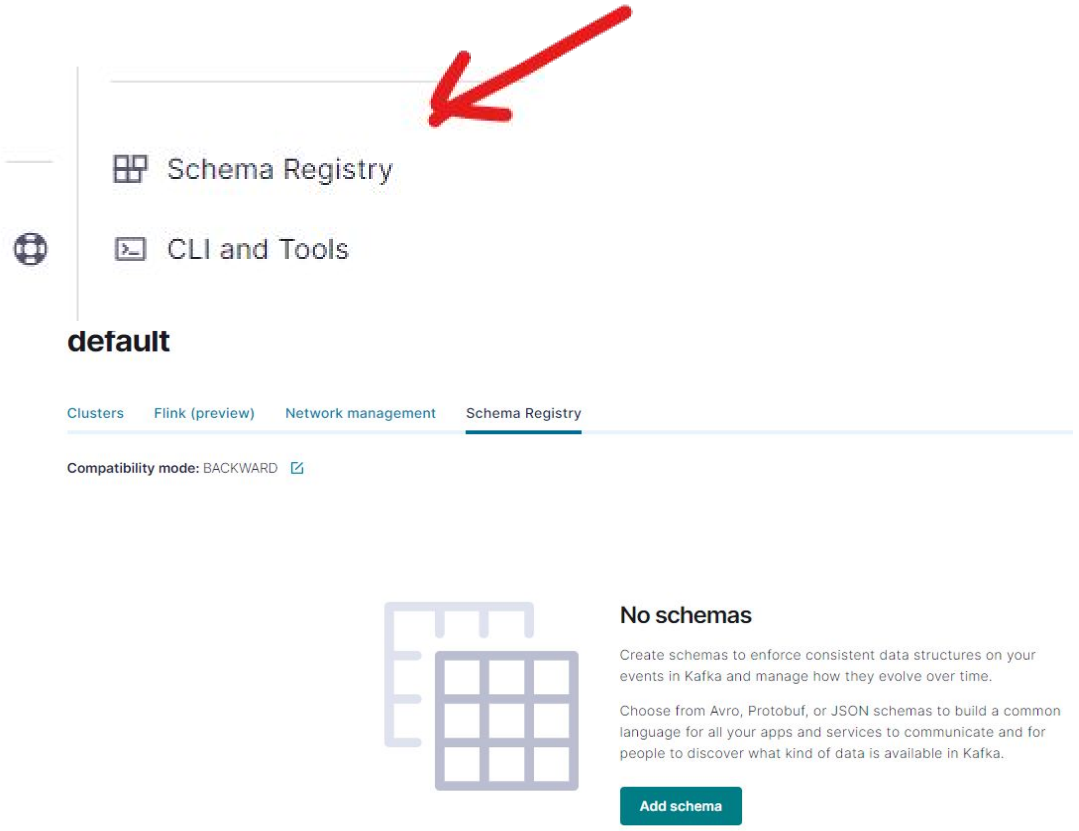
__3. Click on your default cluster.



__4. Click on your cluster, ensure it is running. If not start the cluster.



__5. On the bottom left hand side of the screen menu choose Schema Registry.



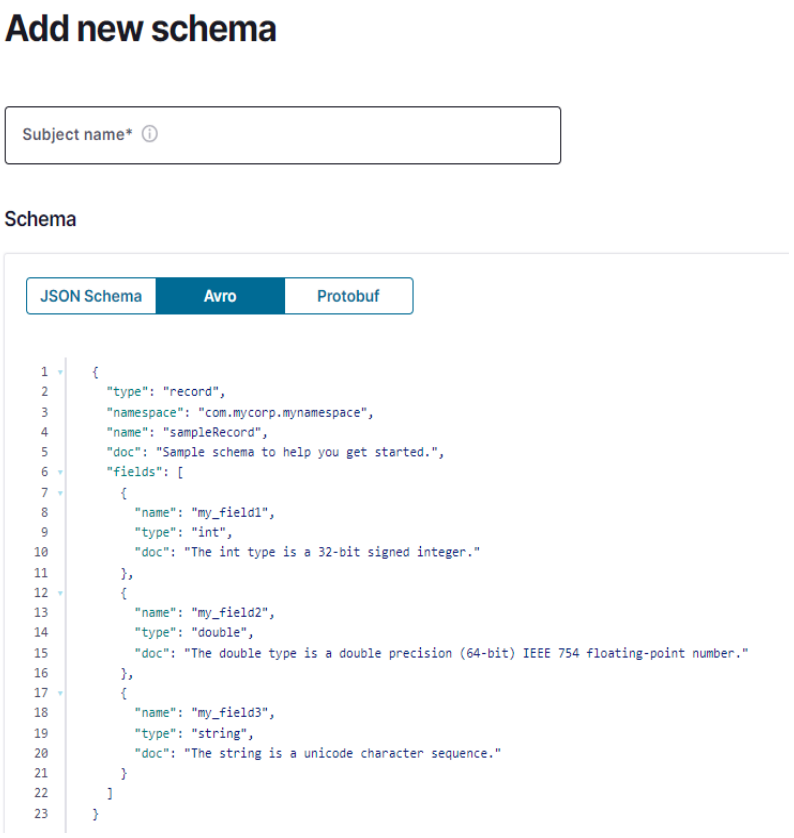
Create schemas to enforce consistent data structures on your events in Kafka and manage how they evolve over time.

Choose from Avro, Protobuf, or JSON schemas to build a common language for all your apps and services to communicate and for people to discover what kind of data is available in Kafka.

__6. This link takes you to the **Add Schema** page within the Confluent Cloud. Click the Add Schema button to begin.

__7. The resulting screen shows example AVRO code and allows the naming of the schema registry to utilize (Subject name). It also shows the choices for Schema Registry types:

JSON, AVRO, or ProtoBuf

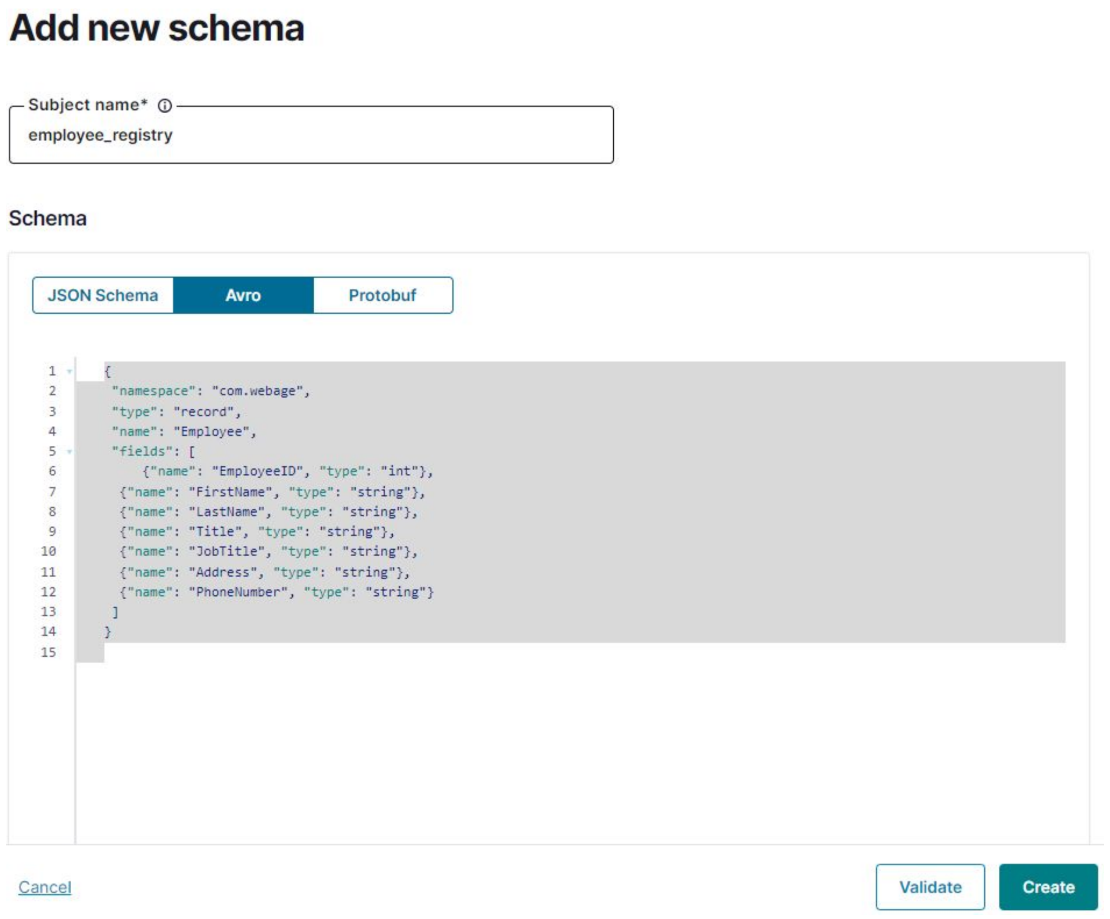


```
{
  "type": "record",
  "namespace": "com.mycorp.mynamespace",
  "name": "sampleRecord",
  "doc": "Sample schema to help you get started.",
  "fields": [
    {
      "name": "my_field1",
      "type": "int",
      "doc": "The int type is a 32-bit signed integer."
    },
    {
      "name": "my_field2",
      "type": "double",
      "doc": "The double type is a double precision (64-bit) IEEE 754 floating-point number."
    },
    {
      "name": "my_field3",
      "type": "string",
      "doc": "The string is a unicode character sequence."
    }
  ]
}
```

__8. Here we will select AVRO as the type and utilize the following code for our **employee_registry** schema name we will create.

```
{
 "namespace": "com.webage",
 "type": "record",
 "name": "Employee",
 "fields": [
     {"name": "EmployeeID", "type": "int"},
  {"name": "FirstName", "type": "string"},
  {"name": "LastName", "type": "string"},
  {"name": "Title", "type": "string"},
  {"name": "JobTitle", "type": "string"},
  {"name": "Address", "type": "string"},
  {"name": "PhoneNumber", "type": "string"}
 ]
}
```

Your final screen should look like below.



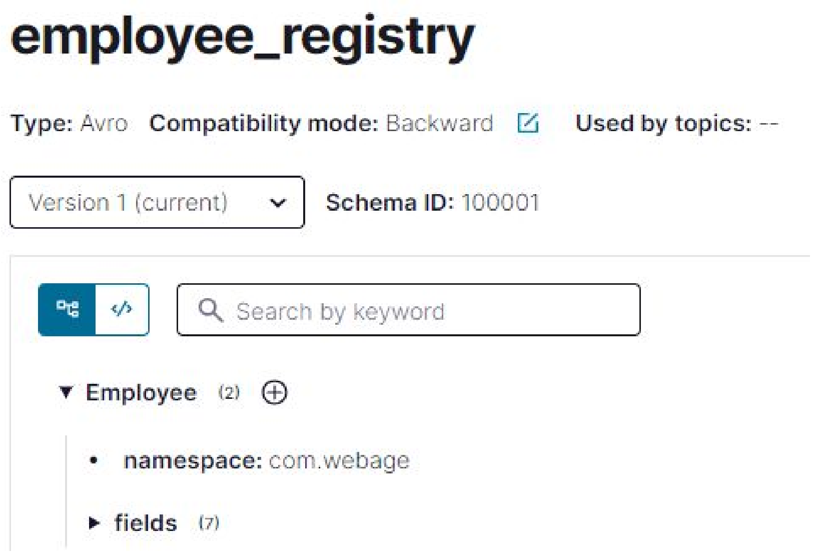
Subject name*: employee_registry

Schema

```
{
 "namespace": "com.webage",
 "type": "record",
 "name": "Employee",
 "fields": [
     {"name": "EmployeeID", "type": "int"},
  {"name": "FirstName", "type": "string"},
  {"name": "LastName", "type": "string"},
  {"name": "Title", "type": "string"},
  {"name": "JobTitle", "type": "string"},
  {"name": "Address", "type": "string"},
  {"name": "PhoneNumber", "type": "string"}
 ]
}
```

__9. Click the **Create** button when ready.

__10. You should be taken to the Schema Registry page to see your new schema listed like below. Notice Confluent automatically sets the version number and compatibility mode. Note on this registry we current have not associated this with any topic – Used by topics


employee_registry

Type: Avro Compatibility mode: Backward Used by topics: --

Version 1 (current) Schema ID: 100001

▼ Employee (2)

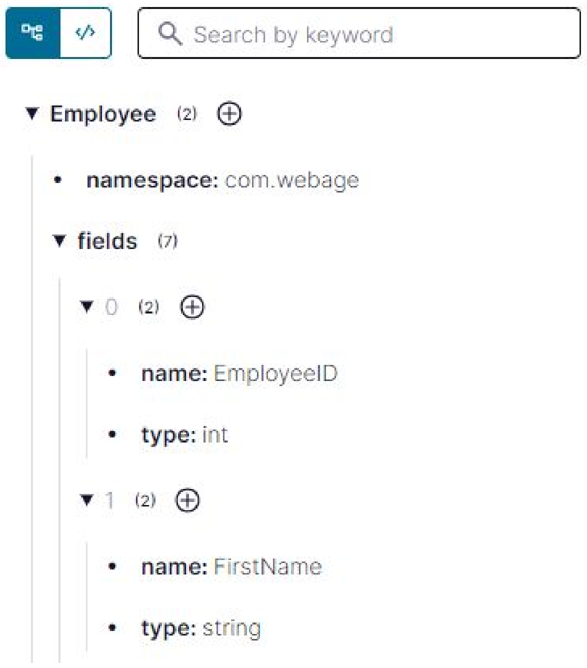
- namespace: com.webage

▶ fields (7)

__11. There are a number of choices for compatibility, click the pen icon as shown above next to Compatibility mode. Here is a list:

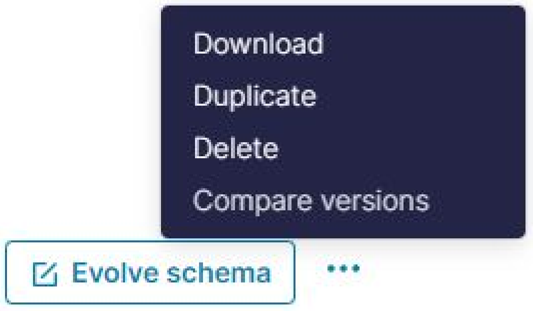
Compatibility Type	Changes allowed	Check against which schemas	Upgrade first
`BACKWARD`	• Delete fields • Add optional fields	Last version	Consumers
`BACKWARD_TRANSITIVE`	• Delete fields • Add optional fields	All previous versions	Consumers
`FORWARD`	• Add fields • Delete optional fields	Last version	Producers
`FORWARD_TRANSITIVE`	• Add fields • Delete optional fields	All previous versions	Producers
`FULL`	• Add optional fields • Delete optional fields	Last version	Any order
`FULL_TRANSITIVE`	• Add optional fields • Delete optional fields	All previous versions	Any order
`NONE`	• All changes are accepted	Compatibility checking disabled	Depends

__12. Expand the fields sections to review the name and data type.


Search by keyword

```
▼ Employee (2)
  • namespace: com.webage
  ▼ fields (7)
    ▼ 0 (2)
      • name: EmployeeID
      • type: int
    ▼ 1 (2)
      • name: FirstName
      • type: string
```

__13. On the right hand side we have some choices available from the three dots next to Evolve Schema. Here you can see the options to Download, Duplicate, Delete and do a diff between versions.

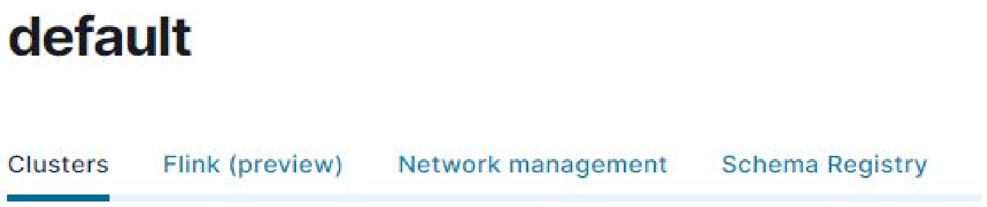


__14. Clicking the Evolve Schema opens an editor window allowing for changes to be made to the current schema set.

__15. The employee_registry schema can now be used when a topic is created in your C# Kafka coding.

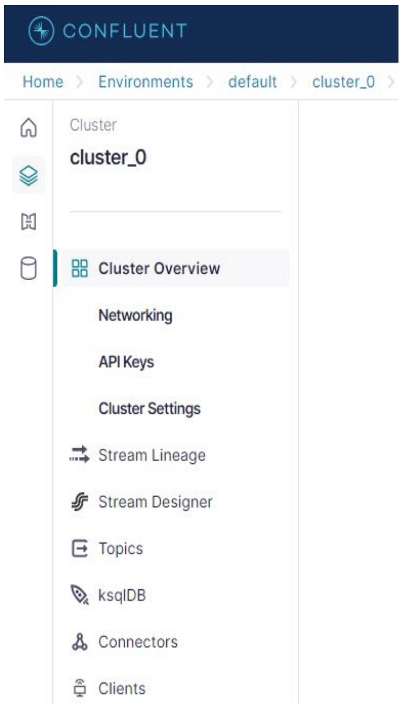
Part 2 - Adding a Schema directly to a Topic

__1. Now lets look at another way to add a schema with a new Kafka topic. Click the **Clusters** tab within the current window.

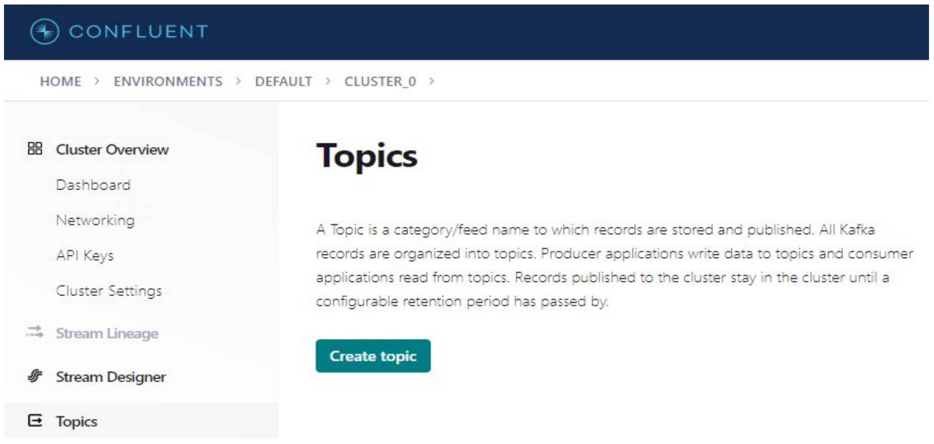


__2. Next click on the default cluster, in this case **cluster_0**, is ours.

__3. On the left-hand menu select the **Topics** link.



__4. If you don't have any Topics, click **Create topic** as shown below, if you already have topics then click **Add topic**.



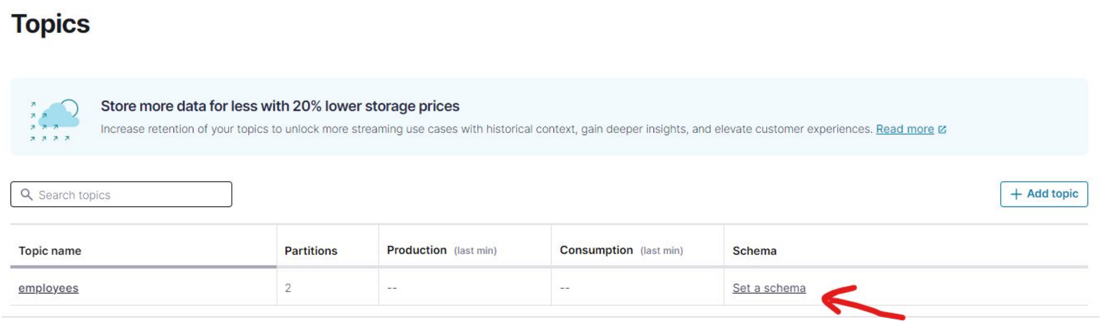
Topics

A Topic is a category/feed name to which records are stored and published. All Kafka records are organized into topics. Producer applications write data to topics and consumer applications read from topics. Records published to the cluster stay in the cluster until a configurable retention period has passed by.

__5. Name the topic ***employees*** and ensure that the Number of partitions is set to **2**. All other items can be left to use the default values. Review the default advanced settings information by clicking the **Show advanced settings** button.

__6. When done, click **Save & create**.

The resulting screen shows the current state of your new topic. Use the tabs to familiarize yourself with the available options here. As you can see, nothing is currently being produced or consumed for the topic orders.



Topics

Store more data for less with 20% lower storage prices

Increase retention of your topics to unlock more streaming use cases with historical context, gain deeper insights, and elevate customer experiences. Read more

Topic name	Partitions	Production (last min)	Consumption (last min)	Schema
employees	2	--	--	Set a schema

__7. Within this window, select the **Set a schema** link under the schema column section, as shown above.

__8. There are two choice here for the Schema: Value and Key

- Set a message value schema to enforce data compatibility.
- Set a message key schema to enforce data compatibility.

A key schema enforces a structure for keys in messages sent to Apache Kafka.

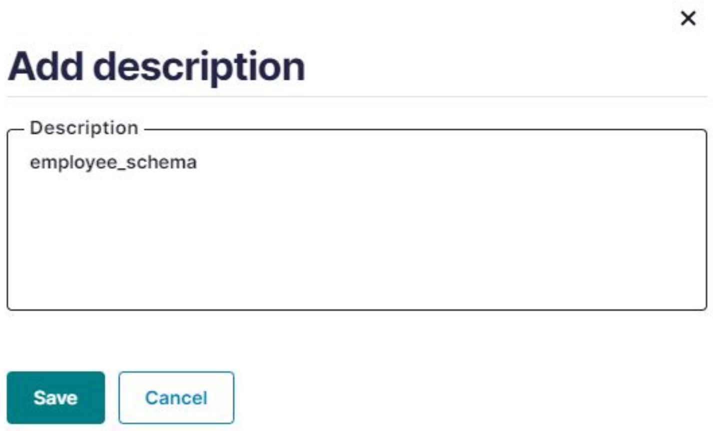
A value schema enforces a structure for values in messages sent to Apache Kafka.

__9. Lets choose Value for this example (default). Click **Set a Schema**.

__10. The Add a new Schema window that opens similar to the one we used previously paste the following AVRO code into the window. Notice we don't set a name here.

```
{
 "namespace": "com.webage",
 "type": "record",
 "name": "Employee",
 "fields": [
     {"name": "EmployeeID", "type": "int"},
   {"name": "FirstName", "type": "string"},
   {"name": "LastName", "type": "string"},
   {"name": "Title", "type": "string"},
   {"name": "JobTitle", "type": "string"},
   {"name": "Address", "type": "string"},
   {"name": "PhoneNumber", "type": "string"}
 ]
}
```

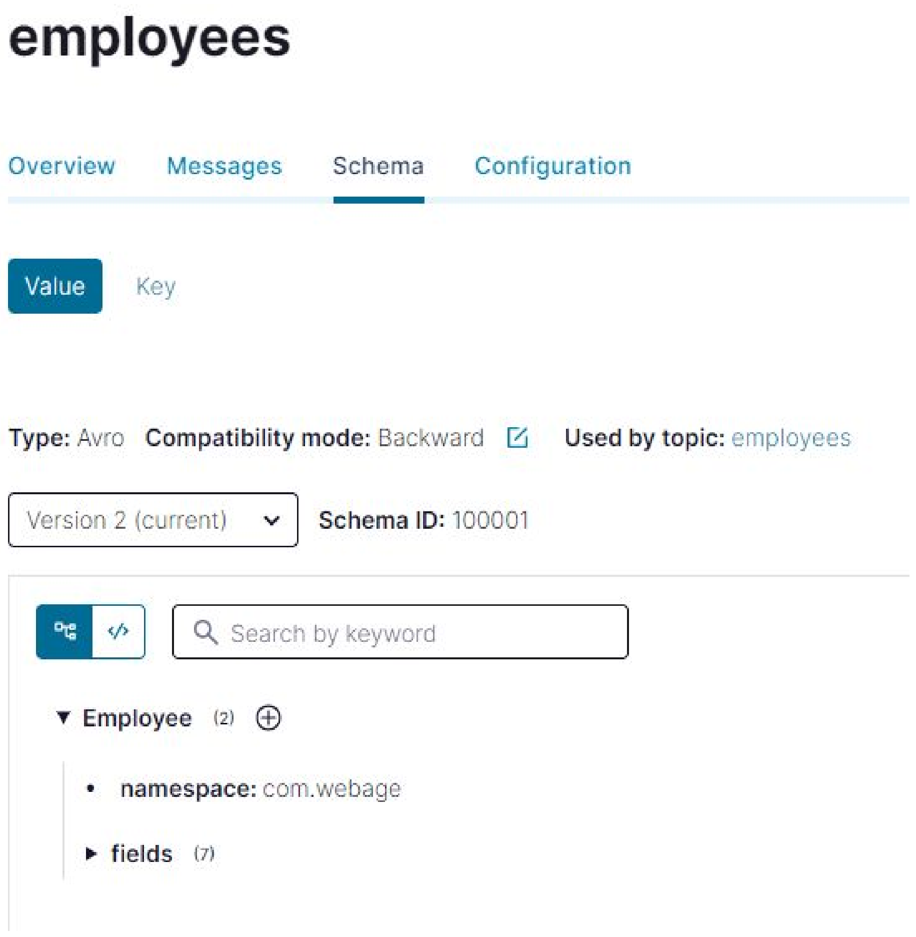
__11. For the description on the far right side add **employees_schema** and click **Save.**



employee_schema

__12. Back in the main window with your code and description showing, click the **Create** button.

__13. You are placed back in the employee schema page associated with the employees topic. It should look similar to the schema registry one from earlier. This one will be called employees. If you look at the current registry for the names of those created because we choose value type, the actual name of the schema is employees_value in Confluent Cloud.



Type: Avro **Compatibility mode:** Backward **Used by topic:** employees

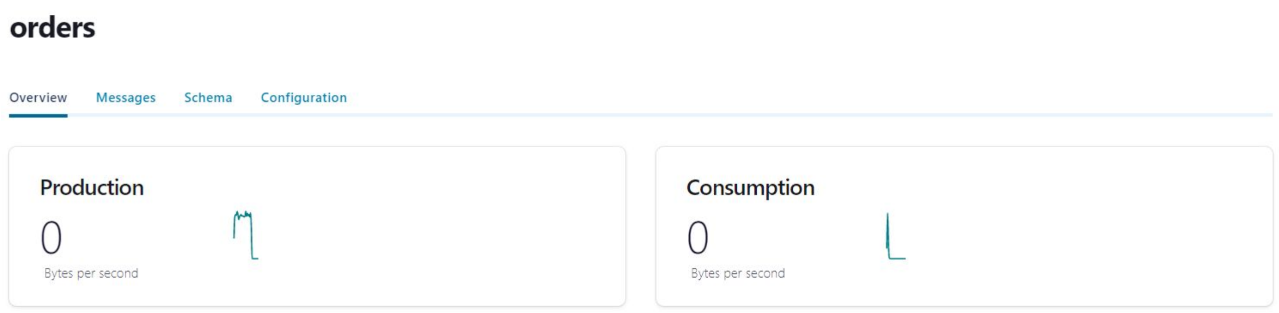
Version 2 (current) **Schema ID:** 100001

Part 3 - Shutdown the Confluent Kafka Connect

__1. Click on **Schema Registry** from the left hand menu and delete the schemas you created. To do so click on the **three dots …** and choose the **Delete** options. You will be prompted to confirm by adding the name of the registry to delete.

__2. Click **Topics** and open the **orders** topic by clicking on it.

You should see something similar to below for your output between Production and Consumption of messages. Again, review the Messages, Schema and Configurations tabs as before.



__3. On the **Configuration** tab, choose the **Delete** topic button to remove the topic.

__4. Complete the information in the popup with topic name **orders** and then click **Continue**.

Part 4 - Review

In this section, you learned the methodology to work with Kafka Schema Registry. We created a schema to be used globally and also worked with a specific topic to add a schema.