

WA3204 Introduction to Kafka for C# Developers

Student Labs

Web Age Solutions Inc.

Table of Contents

Lab 1 - Signing Up for the Free Trial of Confluent Cloud.....	3
Lab 2 - Understanding Confluent Cloud Clusters.....	6
Lab 3 - Understanding Confluent Cloud CLI.....	16
Lab 4 - Understanding Kafka Topics.....	22
Lab 5 - Using the Confluent CLI to Consume Messages.....	36
Lab 6 - Creating an ASP.NET Core MVC Kafka Client.....	38
Lab 7 - Creating an ASP.NET Core WebAPI Kafka Client.....	54
Lab 8 - Creating a .NET Core 3.1 Worker Kafka Client.....	65
Lab 9 - Integrating Azure SQL and Kafka.....	76
Lab 10 - Integrating Redis Cache with .NET Core 3.1 ASP.NET MVC Lab.....	86
Lab 11 - The Final Project.....	98

Lab 1 - Signing Up for the Free Trial of Confluent Cloud

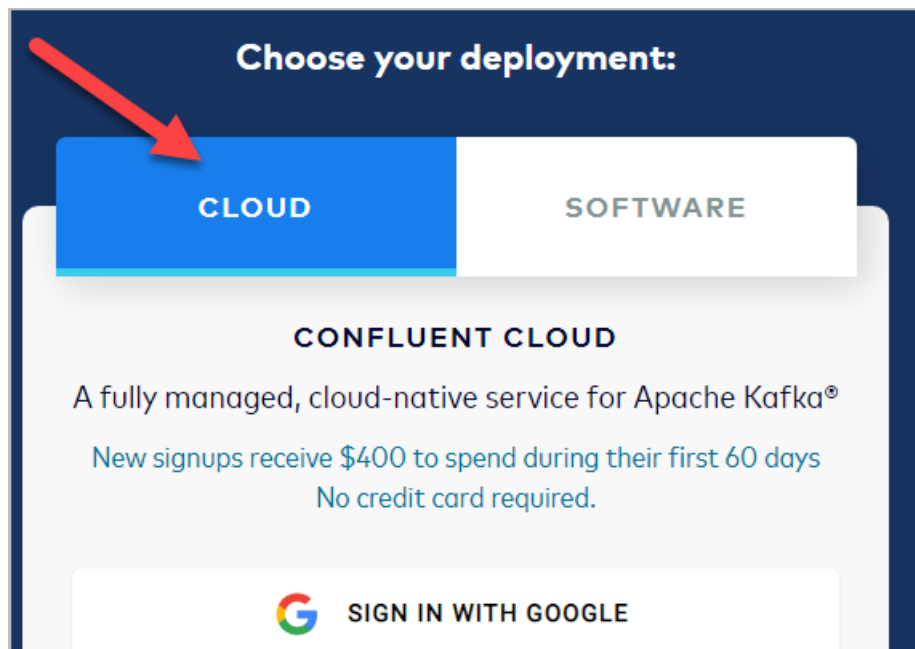
The steps below will guide you through the sign-up process for a free trial of the Confluent Cloud platform. You will have 60 days or \$400 worth of free usage, whichever occurs first.

During the sign-up process, you will be asked to provide your name, some additional information, and a valid email address – no credit card information is required.

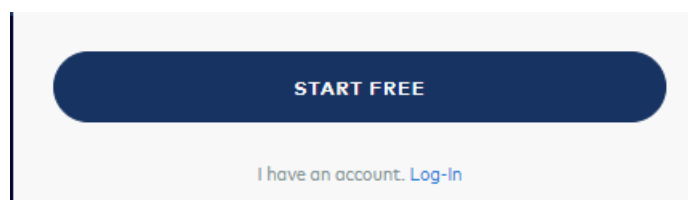
__1. Open your browser and navigate to the Confluent home page

`https://www.confluent.io/get-started/`

__2. Make sure the **CLOUD** development environment tab is selected.



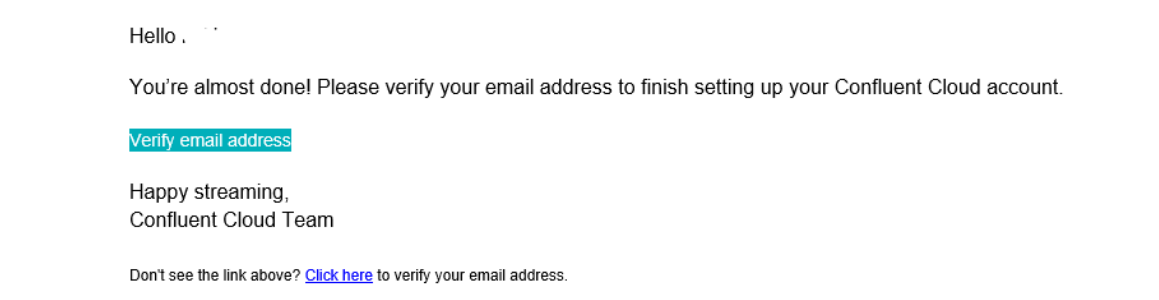
__3. Fill out the form and click **START FREE**



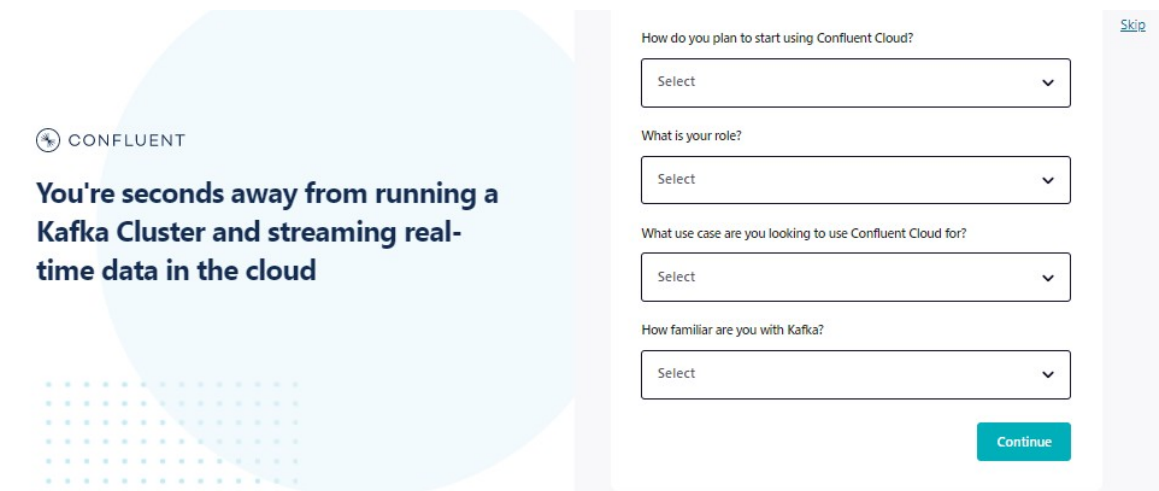
Note: You can also use an option to sign in with Google, or log in, if you already have an account with Confluent.

__4. Click **Accept All Cookies** when prompted.

__5. Open the verification email sent to you and click **Verify email address**.



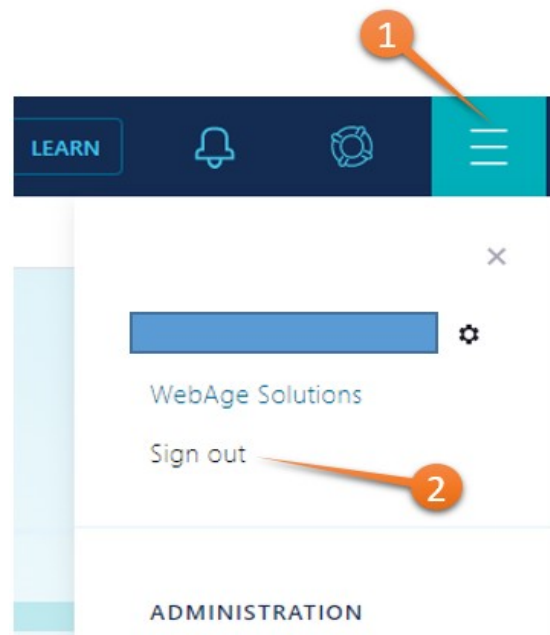
__6. You will be placed on the Confluent welcome page.



__7. Click **Skip** in the upper-right corner.

__8. You will be placed on the *Environments* dashboard page.

__9. Click the "burger" menu in the top right-hand corner, and click **Sign out**

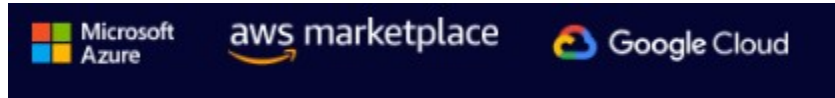


The Confluent Cloud sign-up process is now complete, and you can now log in to Confluent Cloud using your Confluent credentials.

Lab 2 - Understanding Confluent Cloud Clusters

This lab assumes that you have signed up for a (free trial) development account at Confluent Cloud.

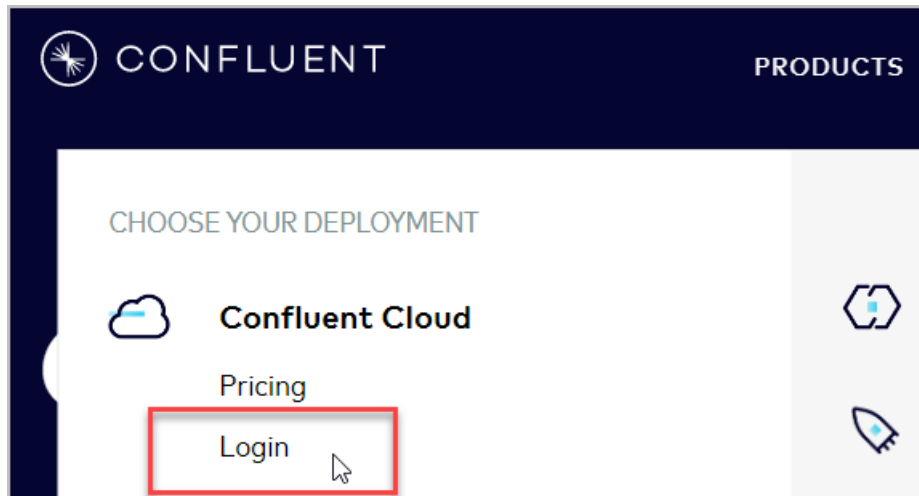
Practically all of the material in this lab is also applicable if you have signed up through your cloud vendor's marketplace to unify billing and leverage existing commitments.



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

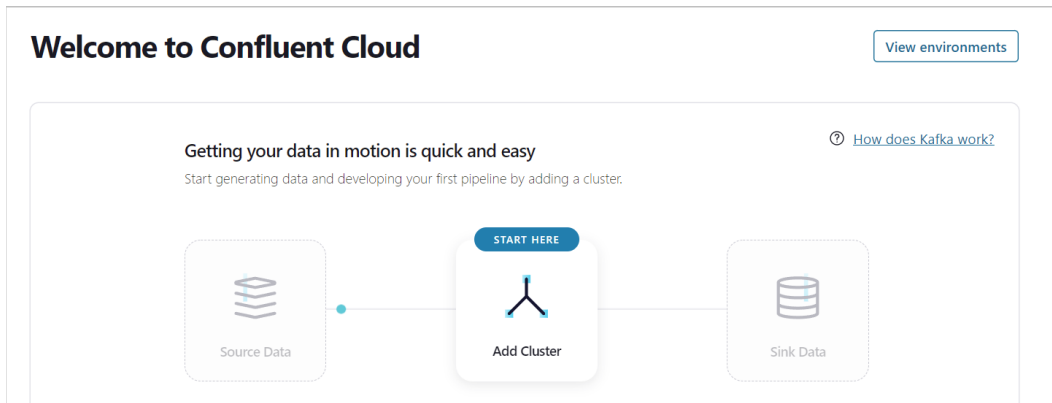
- __ 1. Open your browser and navigate to <https://www.confluent.io/>
- __ 2. Under **PRODUCTS**, select **Confluent cloud** > **Login**



- __ 3. Provide your credentials to log in.
- __ 4. Click **Skip** or otherwise discard any non-essential marketing popups.

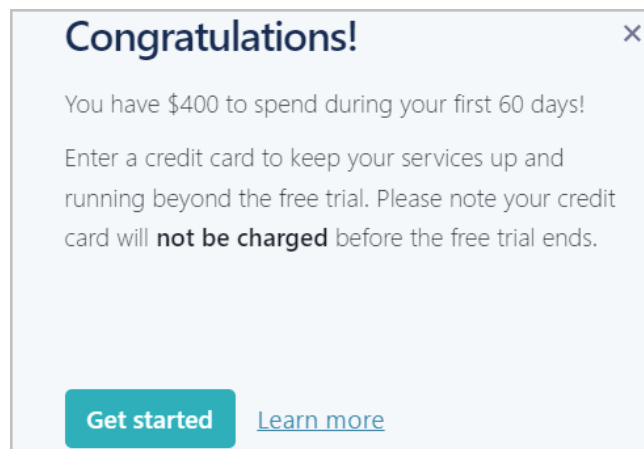
Part 2 - Create a Cluster

__1. You may be presented with the following page, click **Add cluster**




Note, if you see another page, then click **Create cluster on my own**

__2. Then you should see this, click **Get started**



___3. On the **Create cluster** page, click **Begin configuration**

Recommended



Basic

For learning and exploring Kafka and Confluent Cloud.

Ingress	up to 100 MB/s
Egress	up to 100 MB/s
Storage	up to 5,000 GB
Client connections	up to 1,000
Partitions	up to 2,048 (includes 10 free partitions)
Uptime SLA	up to 99.5%

Begin configuration

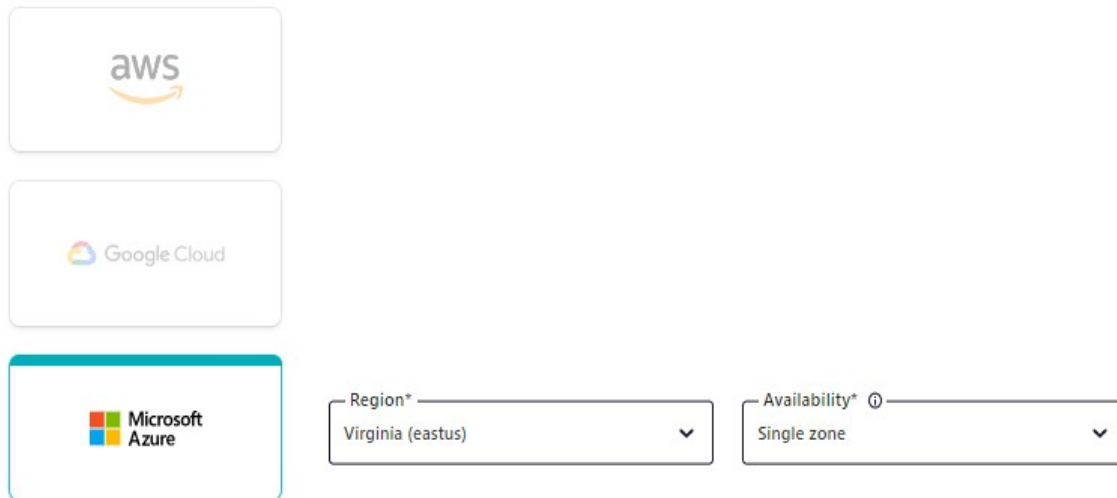
__4. Make sure you select the **Microsoft Azure** provider is selected and configure the following properties:

* For *Region*, select **Virginia (eastus)**

* For *Availability*, select **Single zone**

Create cluster

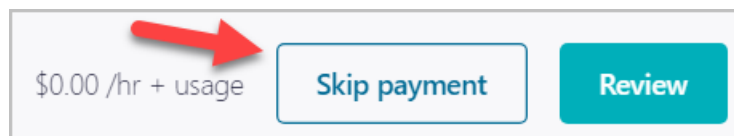
1. Select cluster type — 2. Region/zones — 3. Set payment — 4. Review and launch



The screenshot shows the 'Create cluster' wizard with the 'Region/zones' step selected. Three cloud providers are listed: AWS, Google Cloud, and Microsoft Azure. Microsoft Azure is selected and highlighted with a blue border. To the right of the providers, there are two dropdown menus. The first is labeled 'Region*' and is set to 'Virginia (eastus)'. The second is labeled 'Availability*' and is set to 'Single zone'.

__5. Click **Continue**

__6. Click **Skip payment**



The screenshot shows a payment section with a light gray background. On the left, it says '\$0.00 /hr + usage'. To the right of this text are two buttons: 'Skip payment' (a light blue button with a blue border) and 'Review' (a solid teal button). A red arrow points from the cost text towards the 'Skip payment' button.

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.

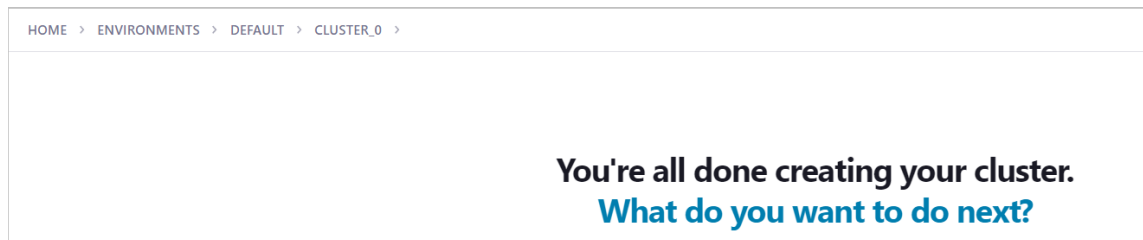
Note: Many programming languages offer functionality for sizing in-memory objects; you can use the available object serialization/marshaling mechanism to estimate the object size to be sent over the network.

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/>)

You can 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.

7. Click **Launch cluster**

The cluster, by default named **CLUSTER_0**, will be provisioned on the selected cloud platform in the **DEFAULT** Confluent environment.

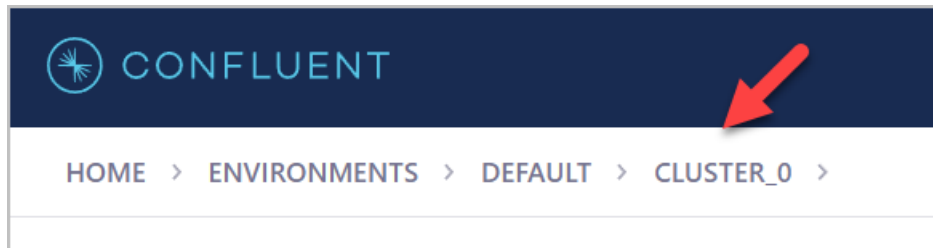


At this point, the **CLUSTER_0** is up and running and you can either set up a Kafka connector to start pumping messages in or out of your cluster, or set up a Kafka client to get low-level programmatic access to Kafka producer and/or consumer APIs.

Part 3 - Cluster Settings Overview

Let's review the **CLUSTER_0** configuration settings.

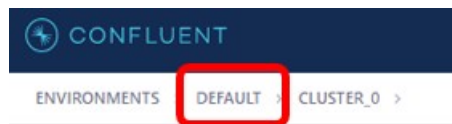
__1. In the navigation breadcrumb path, click **CLUSTER_0**



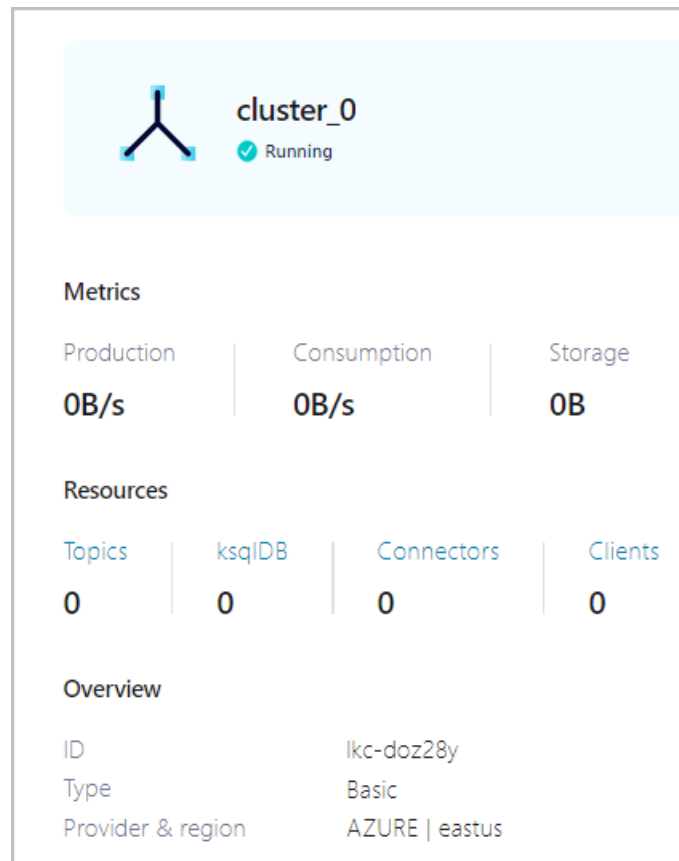
You will be placed on the **Cluster overview** page.

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.)

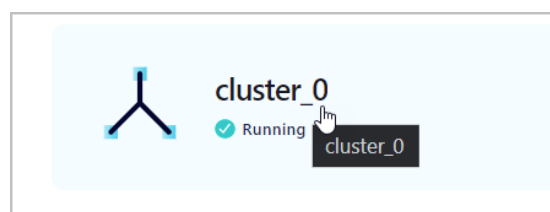
__2. Click the **DEFAULT** environment.



Now you should see that the cluster is marked as running.

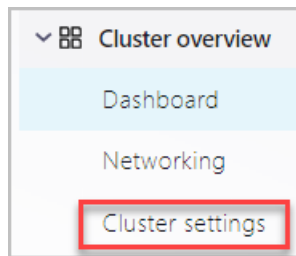


__3. Go back to the cluster overview page by clicking on **CLUSTER_0**



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.

__4. Click **Cluster settings** under **Cluster overview** on the left menu.



You should see a summary of the **cluster_0** settings, including the REST endpoint URI that you can use to interact with your Confluent Cloud using REST-enabled clients.

General

Capacity

Identification

Name

cluster_0

Cluster ID

lkc-doz28y

[🔗](#)

Endpoints

Bootstrap server

pkc-epwny.eastus.azure.confluent.cloud:9092

REST endpoint

https://pkc-epwny.eastus.azure.confluent.cloud:443

🔔

Use the [Kafka REST API](#) to interact with your cluster and produce [records](#) [🔗](#).

Cloud details

Provider

AZURE

Region

eastus

Availability

Single zone

Cluster type

Basic

For basic or development use cases

- Includes 10 free partitions
- [99.5%](#) [🔗](#) uptime SLA
- 5 TB storage

Upgrade available

Enhance your experience

Upgrade to a Standard cluster for production-ready features.

[See details](#)

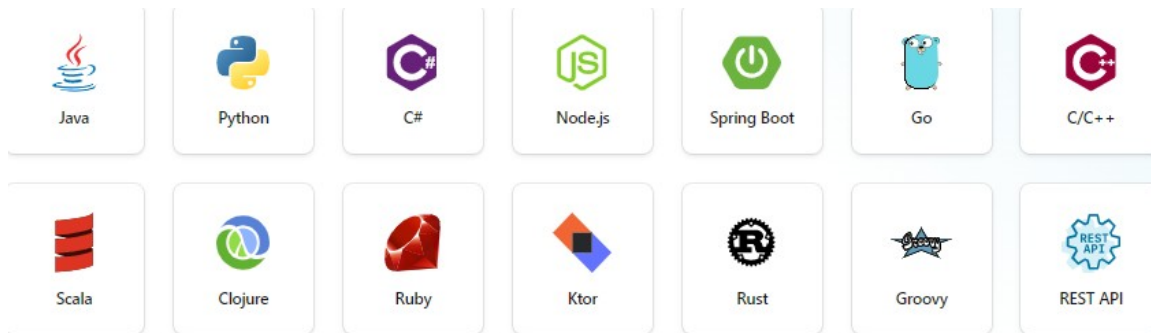
__5. Click **Topics** on the left menu.

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.

__6. Click **Data Integration** on the left menu, and select **Clients** (if it is not selected by default).

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.

__7. Click **Connectors** under **Data Integration**

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 200+ 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.

__ 8. Click **API keys** under **Data Integration**

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 3 - 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 a number of OSes.

__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. 2.19 (the latest at the time of this writing) on Window® 64-bit.

__3. Scroll down and under **Tarball or Zip installation**, click **Windows/x64**

Tarball or Zip installation

1. Download and install the most recently released CLI binaries for your platform:

- [Darwin/x64](#)
- [Windows/x64](#)
- [Linux/x64 \(glibc\)](#)
- [Alpine/x64 \(musl\)](#)
- [Linux/386](#)
- [Checksums \(operating system agnostic\)](#)

The related bundle will get downloaded to your local machine.

For Confluent CLI v. 2.21.0, the binary name is:

```
confluent_v2.21.0_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 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` sub-command).

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

__5. Enter the following command:

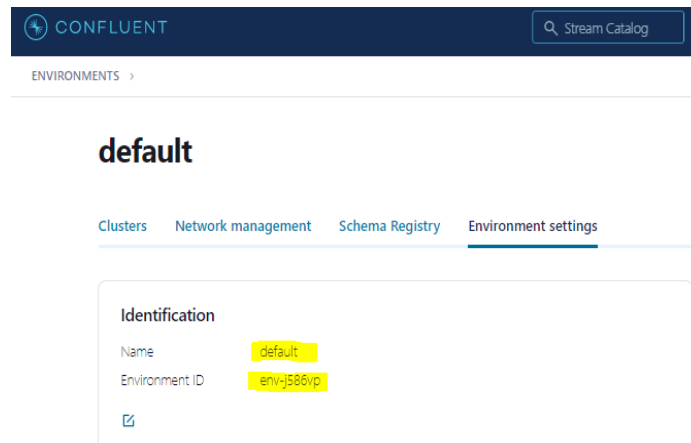
```
confluent environment list
```

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	gcp	us-west4	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 4 - 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.

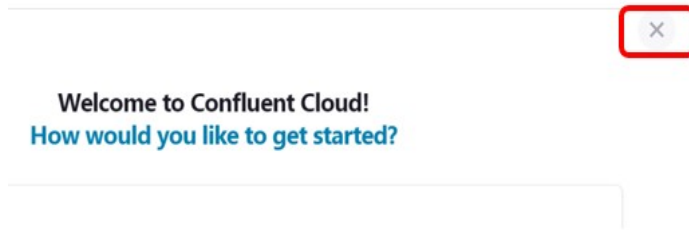
Messages that are consumed from a topic are scanned (think an SQL table scan) starting from a location on a partition (controlled by the offset parameter) all the way to the partition's tail (which can be a moving target with new messages being constantly published to the topic and appended to the partitions), 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.

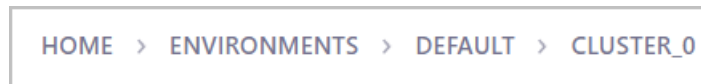
<p>Note: The Confluent Cloud Web UI is constantly evolving and despite our best effort to keep up with the changes, you may not see certain UI elements as they were captured at the time of this writing.</p>

Part 1 - Log in to Confluent Cloud

- __1. If you are signed out, log in back to Confluent Cloud at <https://www.confluent.io/>
- __2. Discard any welcome/marketing pages.



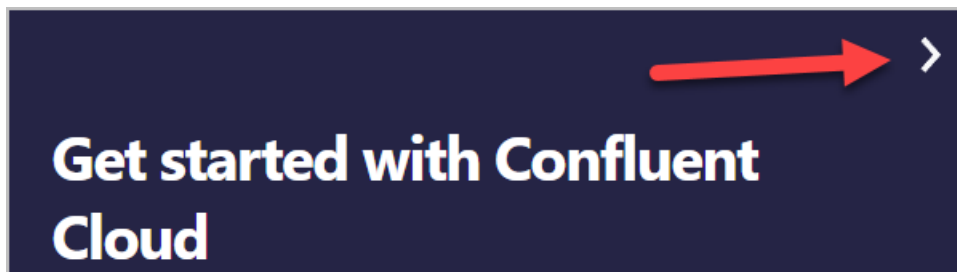
- __3. Navigate to your **CLUSTER_0** overview page on the Cluster Dashboard.
You should see:



- __4. Click **Topics** in the left-hand side navigation bar.

You will be taken to the page titled **New topic** if there are no topics yet or **Topics** if you have already created topics before.

Note. If you see a section with title as Get started with confluent cloud, press the > arrow to compress it.



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:

* For *Topic* name, enter **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.

Storage

Cleanup policy ⓘ Delete	Retention time ⓘ 1 week	Retention size ⓘ Infinite
----------------------------	----------------------------	------------------------------

Message size

Maximum message size in bytes ⓘ 2097164 bytes
--

Other Settings

cluster	cluster_0
replication.factor	3
min.insync.replicas	2
compression.type	producer
leader.replication.throttled.replicas	
message.downconversion.enable	true

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.

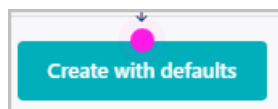
The replication.factor and min.insync.replicas configuration parameters are Kafka's data storage reliability guarantee as replicas are essentially data backups. When the number of replicas falls below min.insync.replicas, the brokers will stop accepting messages and start sending to the producers instances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



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. Click the **Messages** tab.

__2. Click **+ Produce a new message to this topic**

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

```
Value
1  {
2    "ordertime": 1497014222380,
3    "orderid": 18,
4    "itemid": "Item_184",
5    "address": {
6      "city": "Mountain View",
7      "state": "CA",
8      "zipcode": 94041
9    }
10 }
```

```
Key ①
1  18
```

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.

___3. Do 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

The screenshot shows a message editor interface. The top section is labeled "Value" and contains a JSON document. The bottom section is labeled "Key" and contains the value "20". An orange arrow points from the "orderid" field in the JSON document to the "20" in the Key section.

```
Value
1 {
2   "ordertime": 1497014222380,
3   "orderid": 20,
4   "itemid": "Item_184",
5   "address": {
6     "city": "Mountain View",
7     "state": "CA",
8     "zipcode": 94041
9   }
10 }
```

Key ①

1 | 20

___4. Click **Produce**

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

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

Message fields

- topic
- partition
- offset
- timestamp
- timestampType
- headers
- key
- ▼ value
 - ordertime
 - orderid
 - itemid
 - ▼ address
 - city
 - state
 - zipcode

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

```
▼ {"ordertime":1497014222380,"orderid":20,"itemid":"Item_184","address":{"city":"Mountain View","state":"CA","zipcode":94041}}
Partition: 0    Offset: 0    Timestamp: 1656438044379
```

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_partitions*. 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!)

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

__5. 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

__6. Click **Produce**

In our test, the message was placed on Partition 1.

7. 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.

The screenshot displays a Kafka UI interface for producing messages. At the top, there are search filters: "Filter by keyword", "Jump to offset", and "offset". Below these, the "Value" field contains a JSON object:

```
{
  "ordertime": 1497014222380,
  "orderid": 27,
  "itemid": "Item_184",
  "address": {
    "city": "Mountain View",
    "state": "CA",
    "zipcode": 94041
  }
}
```

 The "Key" field contains the value "27". At the bottom right of the production form is a "Produce" button. Below the form, a list of published messages is shown, sorted by "Newest". The list contains four messages with their respective partition, offset, and timestamp:

- Partition: 0 Offset: 4 Timestamp: 16564388931939
- Partition: 0 Offset: 3 Timestamp: 1656438898598
- Partition: 1 Offset: 2 Timestamp: 1656438869213
- Partition: 1 Offset: 1 Timestamp: 1656438849720

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.

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:



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

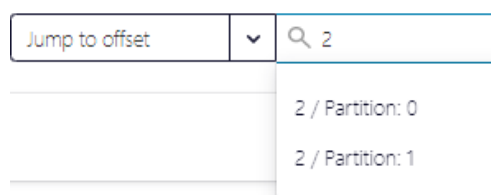
- **Jump to offset,**
- **Jump to time,** or
- **Jump to timestamp**

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

Let's see this in action.

__8. Select **Jump to offset** in the middle window and enter **2** in the second window.

The second window will expand prompting you to select the partition where to do the search by offset:




In our tests, in partition 0, we have three messages from (and including) offset 2 (which is mapped to the 3rd message on the partition) -- it may be different in your case -- and we select that option: **2 / Partition: 0**.

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.

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

10. Select **Jump to the 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 2 messages in our case.



Jump to timestamp

▼

[+ Produce a new message to this topic](#)

▼

```
{"ordertime":1497014222380,"orderid":27,"itemid":"Item_184","address":{"city":"Mountain View","state":"CA","zipcode":94041}}
```

Partition: 0 Offset: 4 Timestamp: 1656438931939

▼

```
{"ordertime":1497014222380,"orderid":26,"itemid":"Item_184","address":{"city":"Mountain View","state":"CA","zipcode":94041}}
```

Partition: 0 Offset: 3 Timestamp: 1656438898598

You can narrow down the search results by providing a (unique) value in the **Filter by keyword** window -- the messages retrieved so far will be scanned for that value -- you could use the **orderid** field's value for that.

The default **cards** view of the search results may not be as informative as the alternative **table** view that you can select by clicking the second part of the composite icon in the upper-right corner:



which, when enabled, will provide a tabular view of the fetched messages:

<input type="checkbox"/>	topic	partition	offset	timestamp	timestampType	headers	key
<input type="checkbox"/>	TestTopic	0	4	1656438931939	CREATE_TIME	[]	27

Part 4 - 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
```

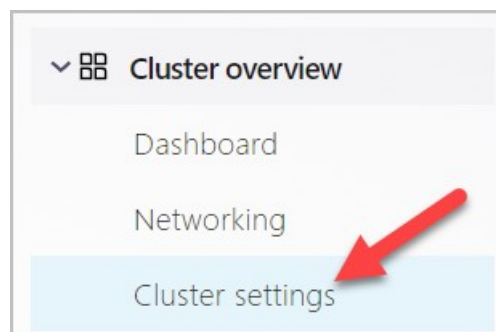
You should see the following output for the available *topic* sub-commands:

```
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.
```

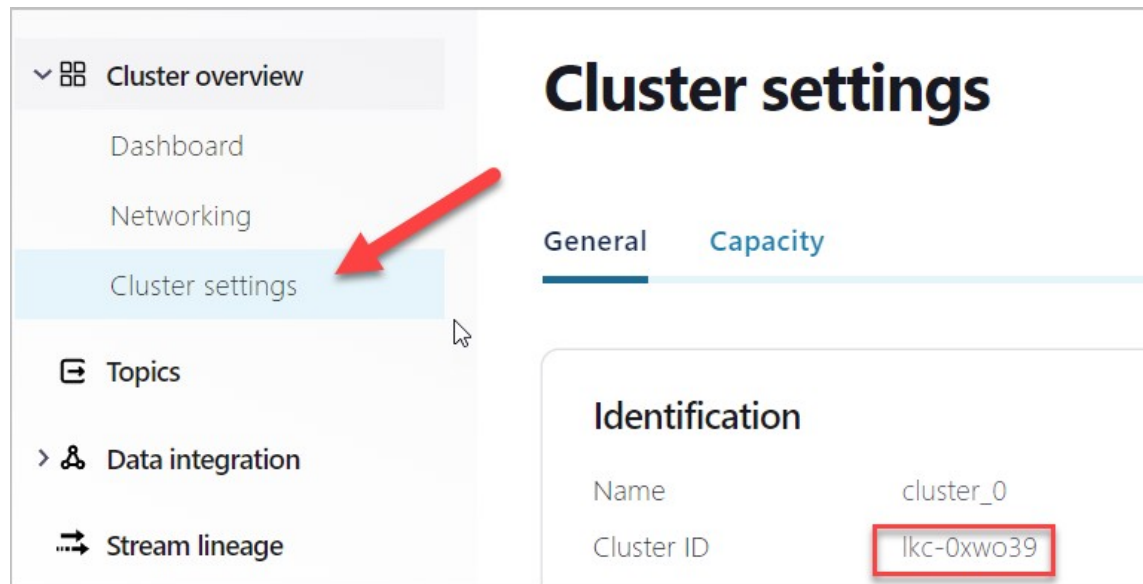
As you can see, with *confluent* you have full control over the lifecycle of a topic, including message consumption (with *confluent* acting as a message consumer).

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. Copy the cluster ID, you will use it in the next command.



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

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

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

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

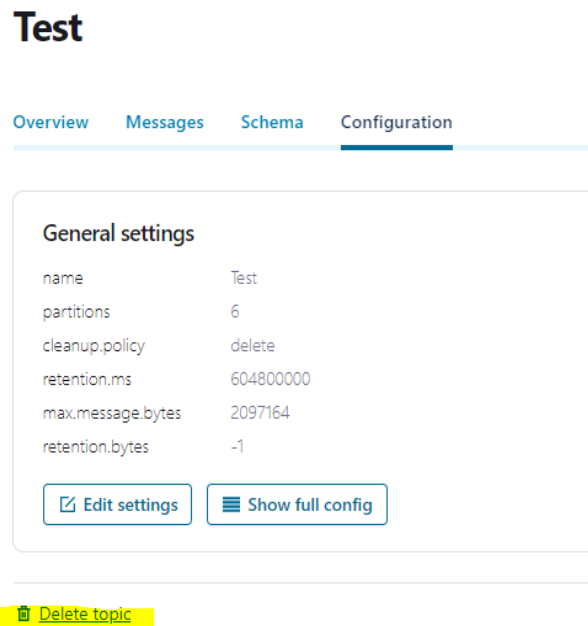
Name	Value
cleanup.policy	delete
compression.type	producer
delete.retention.ms	86400000
file.delete.delay.ms	60000
flush.messages	9223372036854775807
flush.ms	9223372036854775807
follower.replication.throttled.replicas	
index.interval.bytes	4096
leader.replication.throttled.replicas	
max.compaction.lag.ms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

Part 5 - 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 the **TestTopic** topic in this lab.

In the Confluent Cloud portal, click the **Configuration** tab for your topic to see the **Delete topic link** at the bottom of the page as shown below.



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 6 - Review

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

Lab 5 - 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 **TestTopic** topic you created and populated with messages and the *confluent* CLI tool you installed locally in previous labs.

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 command specifying the offset and partition information as well as the API key/secret [in 1 line]:

```
confluent kafka topic consume TestTopic --offset 2 --partition 0 --
cluster <Your cluster Id> --api-key <Your API Key from the above
command> --api-secret <Your Secret>
```

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.

__7. 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 6 - Creating an ASP.NET Core MVC Kafka Client

In this lab, you will add Kafka support to an existing ASP.NET .NET Core MVC 3.1 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:



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

__8. In the terminal window, execute the following command to build the sample application:

```
dotnet build
```

__9. After the build process completes, execute the following command to run the ASP.NET Core MVC application in the built-in webserver:

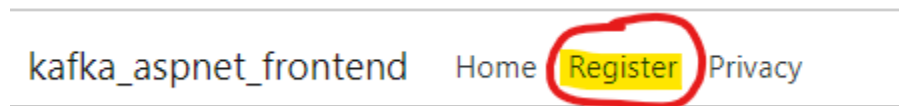
```
dotnet run --urls=http://localhost:8080
```

__10. Open the Chrome browser. If Chrome is not available, you can use Microsoft Edge.

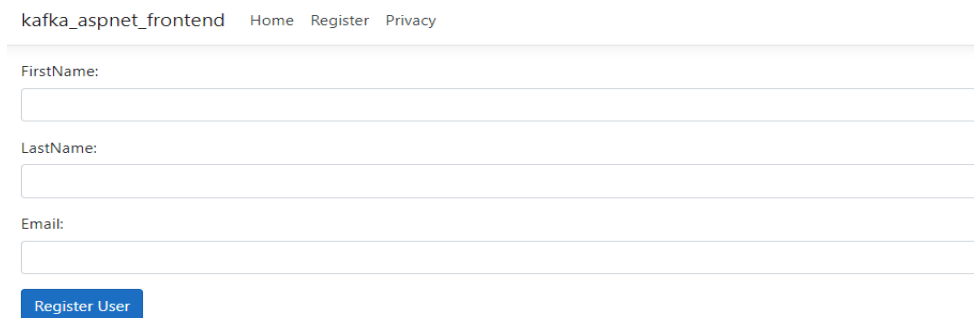
__11. In the browser, navigate to the following URL:

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

__12. On the menu bar, click Register.



Notice the site looks like this:

A screenshot of a web application's registration form. The form has a header with the text 'kafka_aspnet_frontend' and navigation links 'Home', 'Register', and 'Privacy'. Below the header, there are three input fields labeled 'FirstName:', 'LastName:', and 'Email:'. At the bottom of the form is a blue button labeled 'Register User'.

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

__13. In the 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\Home.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] .cloud:443

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

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

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, execute the following commands one at a time to add the required packages to your project:

```
dotnet add package Confluent.Kafka -v 1.9.0
```

```
dotnet add package Newtonsoft.Json -v 13.0.1
```

Confluent.Kafka is required to add the Kafka integration to your project. It contains various classes that helps you to work with topics and also 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="1.9.0" />
  <PackageReference Include="Newtonsoft.Json" Version="13.0.1" />
</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 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):

```
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 shown in **bold**.

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

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

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

ViewBag.Message = "User registration request is sent to the server.";
return View(model);
}
```

The lines in **bold** specifies *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. Once in a while, 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> --api-key <Your API Key> --api-secret <Your Secret>
```

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

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

__4. Switch to the Confluent CLI consumer and notice a message like this shows up: (Note: The values will vary depending on what you entered in the form):

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


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 > ...
1  {
2    "Logging": {
3      "LogLevel": {
4        "Default": "Information",
5        "Microsoft": "Warning",
6        "Microsoft.Hosting.Lifetime": "Information"
7      }
8    },
9    "KafkaConfig": {
10     "BootstrapServers": " ",
11     "SaslUsername": " ",
12     "SaslPassword": " ",
13     "SslCaLocation": "probe"
14   }
15 }
16
17
```

__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 *Startup.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:

```
services.Configure<KafkaConfigModel>(Configuration.GetSection("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 shown in bold:

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

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 - (OPTIONAL) Challenge Exercise - Add support for deleting and creating a topic

In this optional/challenge exercise, you will use the *AdminClient* capabilities to delete and create a Kafka topic. The solution is available in the *C:\LabFiles\solutions* directory.

- ___ 1. Add two buttons to the Register page: (1) Delete Topic (2) Create Topic.
- ___ 2. On **Delete Topic** button click, add the logic to delete the *user-registration* Kafka topic.
- ___ 3. On **Create Topic** button click, add the logic to create the *user-registration* Kafka topic.

Part 10 - Clean-up

- ___ 1. In **VSCode** terminal, press **Ctrl+C** to stop the built-in webserver.
- ___ 2. In the terminal where Confluent CLI consumer is running, press **Ctrl+C**
- ___ 3. Keep the terminal window and **VSCode** running for the next lab.

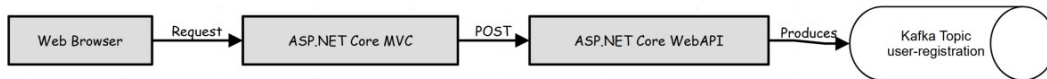
Part 11 - Review

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

Lab 7 - 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*.

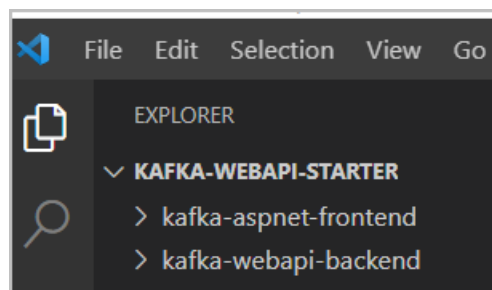
__2. After extraction, ensure the following folders exist.

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 click **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
```

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

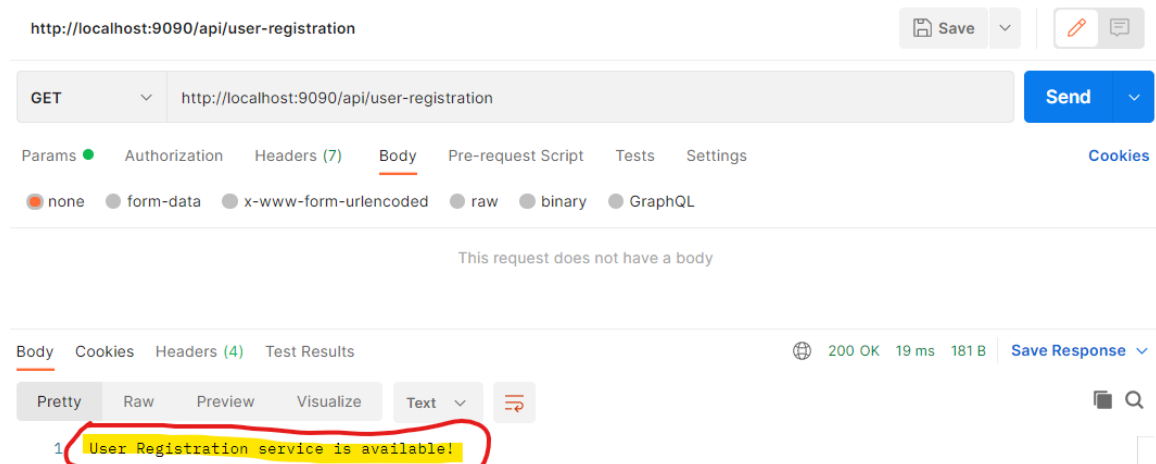
__3. Open **Postman**. If it's not installed in the lab environment, download and install it. It's Free!!!

__4. In **Postman**, ensure the method is set to **GET**.

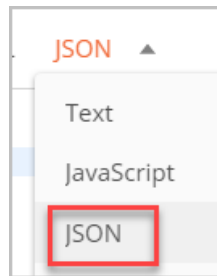
__5. Change URL to **http://localhost:9090/api/user-registration**

__6. Click **Send**.

__7. Ensure it shows the *User Registration Service is available!* in response body.



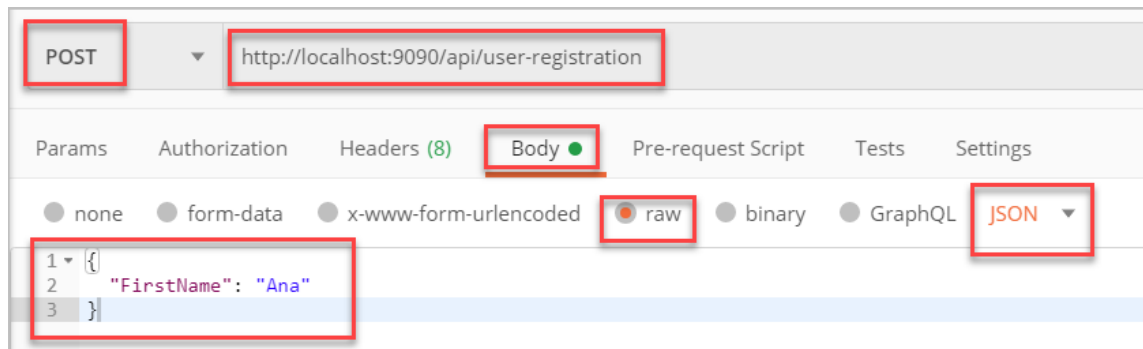
- __ 8. In **Postman**, change method to **POST**
- __ 9. Click the **Body** tab.
- __ 10. Select the **raw** radio button.
- __ 11. Click the drop down and select **JSON**



- __ 12. Enter the following **JSON** fragment. (Note: Don't forget to enter your name.)

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

- __ 13. Ensure your configuration looks like this:



- __ 14. Click **Send**.
- __ 15. Ensure the following message is displayed:

Hello, <Your Name>

- __ 16. Close **Postman** and switch back to **VSCode**.

__17. In **VSCode**, open *kafka-webapi-backend\kafka-webapi-backend.csproj*.

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

```
<PackageReference Include="Confluent.Kafka" Version="1.9.0" />
<PackageReference Include="Newtonsoft.Json" Version="13.0.1" />
```

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.

__18. In **VSCode**, open *kafka-webapi-backend\Models\UserRegistrationModel.cs*.

Notice it is the same file you created in the ASP.NET MVC lab.

__19. Open *kafka-webapi-backend\Controllers\UserRegistrationController.cs*.

__20. Review the Route annotation configured at the class level.

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

__21. 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 3 - Explore the ASP.NET Core MVC Frontend Application

In this part, you will explore the existing frontend 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 note make any changes to the MVC frontend application since the Web API will be responsible for connecting to Kafka.

- ___ 3. In **VSCode**, right click *kafka-aspnet-frontend* and click **Open in Integrated Terminal**.
- ___ 4. Execute the following command in the terminal to launch the built-in web server and host the MVC application.

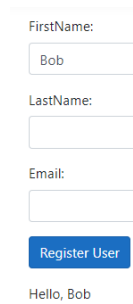
```
dotnet watch run --urls=http://localhost:8080
```

- ___ 5. Open Chrome and navigate to the following URL:

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

- ___ 6. Click **Register**.
- ___ 7. Enter a value in the *First Name* field and click **Register User**.

__ 8. Notice the Hello, <Your name> shows up.



FirstName:
Bob

LastName:

Email:

Register User

Hello, Bob

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

Part 4 - 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:

```
"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\Startup.cs*.

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

__4. In the *ConfigureServices* method, locate the comment
// TODO: add kafka configuration code and add the following code.

```
var producerConfig = new ProducerConfig();  
Configuration.Bind("ProducerConfig", producerConfig);  
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.

__8. Ensure there are no syntax errors in the terminal window where the dotnet watch command is running.

Part 5 - 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 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\UserRegistrationController.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 to the next part.

Part 6 - 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> --api-key <Your API Key> --api-secret <Your API Secret>
```

__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**.

__7. After a moment you will see a message below:

```
Message written: {"UserId":"9a8a65ff-210f-4a61-a86f-db75a468e859","FirstName":"Katy","LastName":"Zeis","Email":"kz@x.com"}
```

__8. Switch to the terminal where the Confluent CLI consumer is running and ensure a message shows up in the terminal.

```
{"UserId":"31775211-559c-4fff-bf1e-7b7112812bd7","FirstName":"Katy","LastName":"Zeis","Email":"kz@x.com"}
```

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

```
KAFKA => Delivered '{"UserId":"9a8a65ff-210f-4a61-a86f-db75a468e859","FirstName":"Katy","LastName":"Zeis","Email":"kz@x.com"}' to 'user-registration [[3]] @0'
```

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 7 - 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.

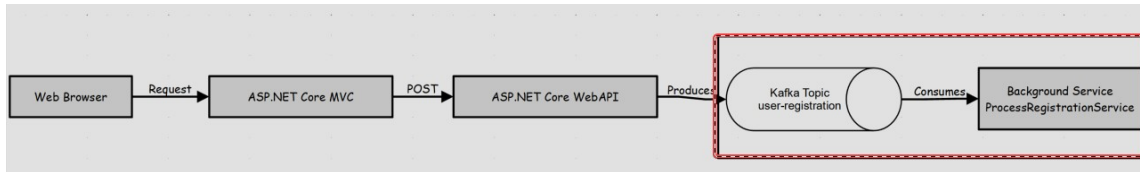
Part 8 - Review

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

Lab 8 - Creating a .NET Core 3.1 Worker Kafka Client

In this lab, you will create a Web API hosted Worker (BackgroundService) 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.

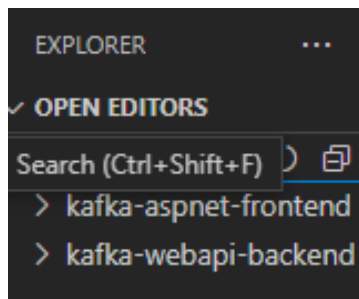


Part 1 - Set-up

__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 Core Web API Project

In this part, you will create a .NET Core 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 Integrated Terminal**.

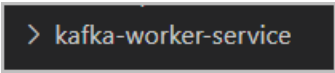
__2. Ensure the terminal shows the following prompt:

```
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
```

It will show in the VSCode explorer.

A screenshot of a VS Code Explorer pane showing a folder named 'kafka-worker-service' with a right-pointing arrow icon next to it.

__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 3.1.27
```

__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 overridden 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\Startup.cs**.

__ 5. Below the existing using statements, add the following namespace shown in bold:

```

...
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Logging;
using kafka_worker_service.Services;

```

__ 6. In Startup.cs, locate the ConfigureServices method and modify it as shown in bold:

```

public void ConfigureServices(IServiceCollection services)
{
    services.AddControllers();
    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.

__7. 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=1.9.0
```

```
dotnet add package Newtonsoft.Json --version=13.0.1
```

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\Startup.cs**

__6. Below the existing using statements, add the following namespace shown in **bold**:

```
...
using kafka_worker_service.Services;
using Confluent.Kafka;
```

__7. Locate the ConfigureServices method and modify it as shown in **bold**:

```
public void ConfigureServices(IServiceCollection services)
{
  services.AddControllers();

  var consumerConfig = new ConsumerConfig();

  Configuration.Bind("ConsumerConfig", consumerConfig);

  services.AddSingleton<ConsumerConfig>(consumerConfig);

  services.AddSingleton<IHostedService, UserRegistrationWorker>();
}
```

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

__ 8. Right click **kafka-worker-service** and create a new folder named **Helpers**

__ 9. Using File Explorer, navigate to **C:\LabFiles\kafka-worker-snippets** and drag & drop **ConsumerHelper.cs** file to VSCode **kafka-worker-service\Helpers** folder.

__ 10. 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.

__ 11. In VSCode, open **Services\UserRegistrationWorker.cs**

__12. 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;
```

__13. 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...");

    while (!stoppingToken.IsCancellationRequested)
    {
        // TODO: worker logic
        var consumerHelper = new ConsumerHelper(_config, "user-
registration");
        string message = consumerHelper.ReadMessage();

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

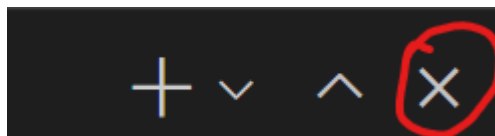
        //TODO:: Process Order
        Console.WriteLine($"---Info: User Registered: {message}---");
    }
}
```

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

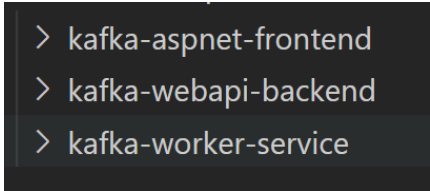
In this lab, you created a .NET Core worker project that act as a Kafka client and consumed messages from the user-registration Kafka topic.

Lab 9 - Integrating Azure SQL and Kafka

In this lab, you will continue with the previous lab and add Azure SQL 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-azure-starter.zip** to **C:\LabWorks\kafka-azure-starter**
- __2. Open VSCode and from the menu click **File** → **Open Folder** and select **C:\LabWorks\kafka-azure-starter**
- __3. Ensure it has the following three projects:



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

Part 2 - Create a Table in Azure SQL Database

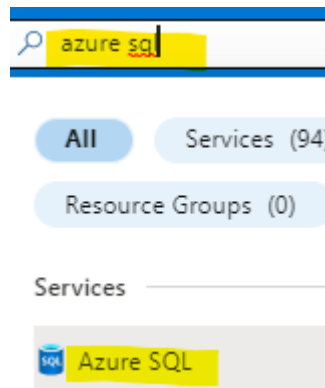
In this part, you will create UserRegistration table in a provided Azure SQL database.

- __1. Using Chrome, navigate to the following URL:

`portal.azure.com`

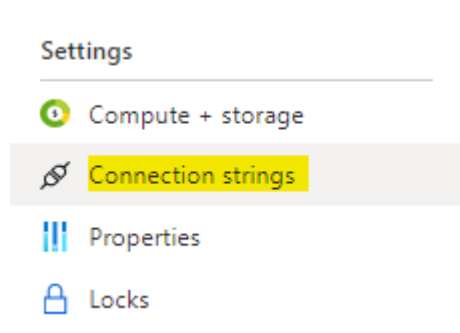
- __2. Login with the Azure credentials that were provided to you as part of the training.
- __3. You may see a message when you login for the first time, click to wait for 14 days.

__4. In the search bar on the top, search for Azure SQL and click on it.



__5. Click the *KafkaLab* database.

__6. In the navigation menu on the left hand side, click **Connection strings**.

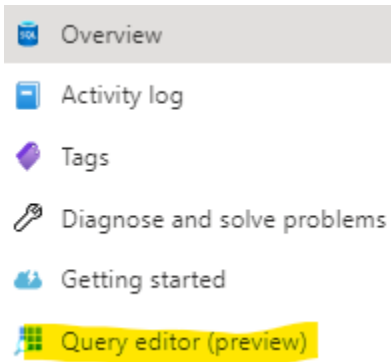


__7. Copy the connection string value and save it in notepad. You will use it later in the lab. The connection string looks like this:

```
Server=tcp:dbsqlserver2022.database.windows.net,1433;Initial  
Catalog=KafkaLab;Persist Security Info=False;User  
ID=dba;Password={your_password};MultipleActiveResultSets=False;Encrypt=  
True;TrustServerCertificate=False;Connection Timeout=30;
```

__8. Change {your_password} to the Azure SQL password that was provided to you as part of the training.

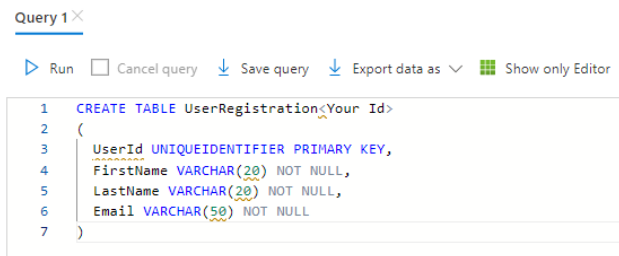
__9. In the navigation menu on the left hand side, click **Query editor**.



__10. Login with the credentials that were provided to you.

__11. Using File Explorer, navigate to **C:\LabFiles\kafka-azure-snippets** and copy the contents of **user-registration-query.txt** to clipboard.

__12. Switch to the Chrome browser where the Query editor page is open and paste the contents in the editor.



__13. Change **<Your Id>** to the student id that was assigned to you as part of the training. It would be a number, such as **01, 02, 03,** The final query should look like this: (Note: Replace **<Your Student Id>** with your ID, for example *UserRegistration001*):

```
CREATE TABLE UserRegistration<Your Student Id>
(
    UserId UNIQUEIDENTIFIER PRIMARY KEY,
    FirstName VARCHAR(20) NOT NULL,
    LastName VARCHAR(20) NOT NULL,
    Email VARCHAR(50) NOT NULL
)
```

Note: Why do you need the id? Because, all participants are sharing the same Azure SQL database and everyone will create the UserRegistration table in the same database. Just so that there's no clash, you are adding a unique id to the table name.

- __14. Click the **Run** button to execute the query.
- __15. Refresh tables in the query editor and ensure the table exists.

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

In this part, you will configure the worker project you developed in the previous lab to support Azure 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 3.1.27
```

```
dotnet add package Microsoft.EntityFrameworkCore.Design --version 3.1.27
```

```
dotnet tool install --global dotnet-ef --version 3.1.27
```

- __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  
3.1.27
```

__5. Execute the following command to create scaffolding on the KafkaLab sample database.

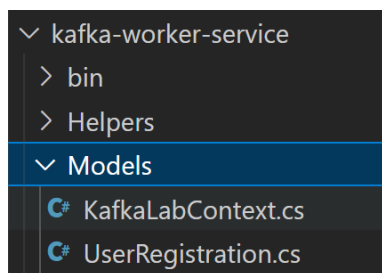
(Note: Don't forget to plug in your connection string and also change --table UserRegistration<Your Student Id> to the id that is assigned to you. The --table part should look like this: --table UserRegistration001 if your id is 001.):

```
dotnet ef dbcontext scaffold "<Your Azure SQL Connection String>"
Microsoft.EntityFrameworkCore.SqlServer --table UserRegistration<Your
Student Id> --output-dir Models
```

The above command connects to the Azure SQL KafkaLab database and generates classes for the database and the UserRegistration table in the custom Models folder.

TROUBLESHOOTING: If the command times out, execute the command again.

__6. Ensure the **Models** folder is created with the following files:



KafkaLabContext.cs contains the KafkaLab database context that will allow you to perform insert, update, delete, and select operations. UserRegistration.cs is mapped to the UserRegistration table in the Azure SQL database.

__7. 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 Azure 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 Azure SQL database table.

__ 1. In VSCode, open **kafka-worker-service\Services\UserRegistrationWorker.cs**.

__ 2. Below the existing using statements, add the following namespace:

```
using kafka_worker_service.Models;
```

__ 3. In the **ExecuteAsync** method, locate the comment
// TODO: add database code here and add the following code:

```
using(KafkaLabContext context = new KafkaLabContext())
{
    UserRegistration<Your Student Id> user =
    JsonConvert.DeserializeObject<UserRegistration<Your Student
    Id>>(message);

    context.UserRegistration<Your Student Id>.Add(user);

    context.SaveChanges();

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

Note: The code uses the context class to connect to Azure SQL database, consumes a message from the Kafka topic, and adds it to the database table.

Your code should look like this example but with your user information:

```
// TODO: add database code here
using(KafkaLabContext context = new KafkaLabContext())
{
    UserRegistration001 user = JsonConvert.DeserializeObject<UserRegistration001>(message);
    context.UserRegistration001.Add(user);
    context.SaveChanges();
    Console.WriteLine($"---Info: User Registered: {message}---");
}
```



__ 4. Open **kafka-webapi-backend\appsettings.json**

__ 5. 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"
}
```

__ 6. Open **kafka-worker-service\appsettings.json**

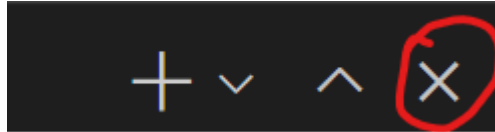
__ 7. Update the placeholders with values you used in the previous labs:

```
},
"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 Azure 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 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'
```

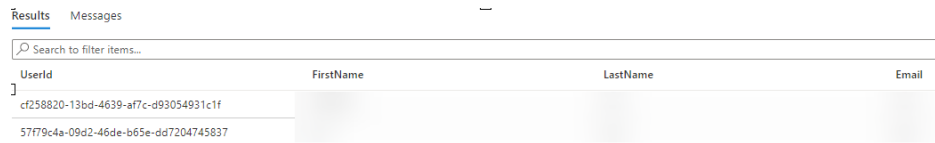
TROUBLESHOOTING: If the worker project doesn't consume and show messages right away, wait for up to a minute so it can catch up with the Kafka topic.

__ 13. Switch to the Chrome browse window where you have Azure Sql **Query editor** open. If the Azure session has timed out, login in to the portal again and navigate to Azure Sql Query editor.

__ 14. In the query editor, execute the following command to retrieve all records from the UserRegistration table. (Note: Don't forget to replace <Your student id> with the value you used earlier in the lab):

```
SELECT * FROM UserRegistration<Your student id>
```

Notice the output looks like this: We have concealed the actual values in the screenshot.



UserId	FirstName	LastName	Email
cf258820-13bd-4639-af7c-d93054931c1f			
57f79c4a-09d2-46de-b65e-dd7204745837			

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.

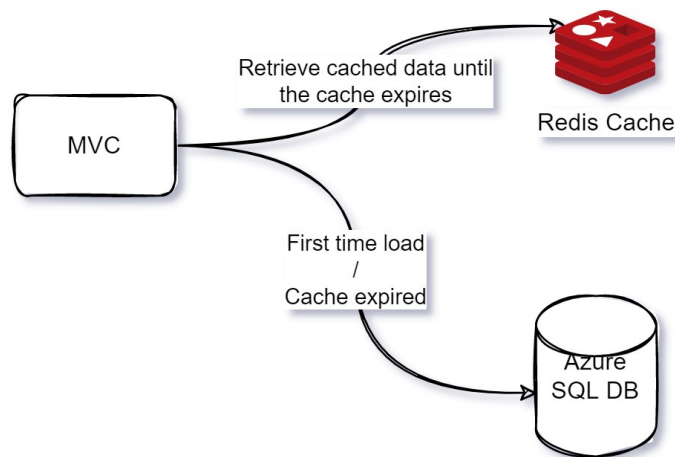
Part 7 - Review

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

Lab 10 - Integrating Redis Cache with .NET Core 3.1 ASP.NET MVC Lab

In this lab, you will integrate Redis Cache with .NET Core 3.1 ASP.NET MVC.

Here is the overall architecture.



You will perform the following operations in the lab:

- Add Entity Framework support to an MVC application.
- Write logic to connect to Azure SQL database to insert and retrieve data.
- Test the application without caching data.
- Add the Redis cache support to the MVC application and cache data for one minute.
- Test the application after enabling caching.

Part 1 - Setup

In this part, you will open a starter MVC project to which you will add the Redis Cache support.

1. Using File Explorer, navigate to **C:\LabFiles\redis-cache-lab-starter.zip** and extract it to **C:\LabWorks\redis-cache-lab-starter**
2. Ensure **C:\LabWorks\redis-cache-lab-starter** exists.

__ 3. Open VSCode.

__ 4. From the menu, click **File** → **Open** and select **C:\LabWorks\redis-cache-lab**

Take a moment to review Views\Home\Register.cshtml and Views\Home\Users.cshtml files. The file contents are commented out. You will modify the code later in the lab. These have been customized for you as part of the starter project. Register.cshtml is the same file you saw in the MVC frontend project. it will let you add more users to the system. Users.cshtml shows the list of existing users.

__ 5. In VSCode, click **Terminal** → **New Terminal** to open a new terminal window.

Part 2 - Add Entity Framework to the MVC Project

In this part, you will add the Entity Framework support to the MVC project.

__ 1. In the VSCode terminal window, execute the following commands to add the Entity Framework support to the project:

```
dotnet add package Microsoft.EntityFrameworkCore.SqlServer --version 3.1.27
```

```
dotnet add package Microsoft.EntityFrameworkCore.Design --version 3.1.27
```

```
dotnet tool install --global dotnet-ef --version 3.1.27
```

Note: The last command might error out saying that dotnet-ef is already installed. Ignore the error.

__2. Execute the following command to create a scaffolding on the Azure SQL table that you utilized in the previous lab.

(Note: Don't forget to plug in your connection string and also change --table UserRegistration<Your Student Id> to the id that is assigned to you. The --table part should look like this: --table UserRegistration001 if your id is 001.):

```
dotnet ef dbcontext scaffold "<Your Azure SQL Connection String>"  
Microsoft.EntityFrameworkCore.SqlServer --table UserRegistration<Your  
Student Id> --output-dir Models
```

__3. Ensure the following files are created in the **Models** folder:

```
KafkaLabContext.cs  
UserRegistration<Your Student Id>.cs
```

Part 3 - Implement the Register POST Logic

In this part, you will implement the user registration POST logic so records can be added to the Azure SQL database.

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

__2. Locate the comment **// TODO: Add the Register() Post method** and add the following code, make sure to update the **<Your Student Id>** with your ID:

```
// TODO: Add the Register() Post method
[HttpPost]
public IActionResult Register(UserRegistration<Your Student Id>
model)
{
    try
    {
        using(var context = new KafkaLabContext())
        {
            model.UserId = Guid.NewGuid();
            context.UserRegistration<Your Student
Id>.Add(model);
            context.SaveChanges();
            ViewBag.Message = "User registered!";
        }
    }
    catch (Exception ex)
    {
        ViewBag.Message = ex.Message;
    }

    return View();
}
```

__3. In VSCode, open **Views\Home\Register.cshtml** and **uncomment** the code by removing **@*** and ***@**. (Note: Do not remove the actual code, only uncomment it.)

__4. Update the first line with your **<Your Student Id>**:

```
@model redis_cache_lab.Models.UserRegistration<Your Student Id>
```

__5. In the terminal window, execute the following command to ensure there are no syntax errors:

```
dotnet build
```

Part 4 - Implement the GET method to retrieve the user list

In this part, you will add the logic to retrieve the user list from the Azure SQL table without involve Redis Cache. You will add the Redis Cache functionality later in the lab.

__ 1. In VSCode, open **Controllers\HomeController.cs**

__ 2. Locate the Users method and modify it as shown in **bold**, make sure to update the **<Your Student Id>** with your ID:

```
public IActionResult Users()  
{  
    var context = new KafkaLabContext();  
  
    var users = context.UserRegistration<Your Student  
Id>.ToList();  
  
    ViewBag.Message = "Getting Data from Azure SQL";  
  
    return View(users);  
}
```

__ 3. In VSCode, open **Views\Users.cshtml** and **uncomment** the code. (Note: Only remove the @* and *@ symbols. Do not remove the actual code.)

__ 4. Update the first line with your **<Your Student Id>**:

```
@model IEnumerable<redis_cache_lab.Models.UserRegistration<Your Student  
Id>>
```

__ 5. In the terminal window, execute the following code and ensure there are no errors:

```
dotnet build
```

Part 5 - Test the application without Redis Cache

In this part, you will test the application without Redis Cache. It will connect to the Azure SQL database each time you refresh the page.

__1. In the terminal window, execute the following command to start up the application:

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

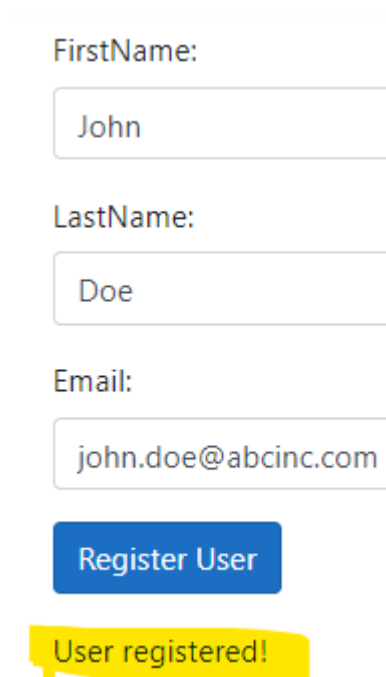
__2. Open Chrome and navigate to the following URL:

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

__3. Click **Register**.

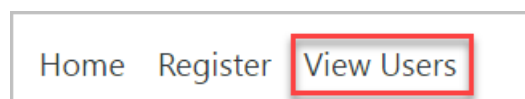
__4. Enter new values (do not use previous value as they are already in the database) in to the fields and click the **Register User** button.

__5. Ensure the page shows the following message:



The screenshot shows a registration form with three input fields: 'FirstName:' containing 'John', 'LastName:' containing 'Doe', and 'Email:' containing 'john.doe@abcinc.com'. Below the fields is a blue button labeled 'Register User'. At the bottom, a yellow message box displays 'User registered!'.

__6. Click **View Users**.



The screenshot shows a navigation bar with three links: 'Home', 'Register', and 'View Users'. The 'View Users' link is highlighted with a red rectangular border.

Notice the user list shows up like this:

User Id	First Name	Last Name	Email
68510571-0ef0-4312-9e41-62df85aed931	John	Doe	john.doe@abcinc.com
3fed3c1b-496a-41dd-82eb-874ce1d9779a	Kat	Ze	kz@x.com

Getting Data from Azure SQL

Also, notice the following message is displayed:

Getting Data from Azure SQL

__7. Try refreshing the page in the browser a few times. You will see the same message that means you are hitting the Azure SQL database each time.

__8. In the terminal window, press Ctrl+C to stop the application.

Part 6 - Obtain Redis Cache Connection Details

In this part, you will connect to the Azure portal and obtain information needed to connect to Redis cache.

__1. Using Chrome or Edge, navigate to the following URL:


`portal.azure.com`

__2. Sign in with your assigned credentials.

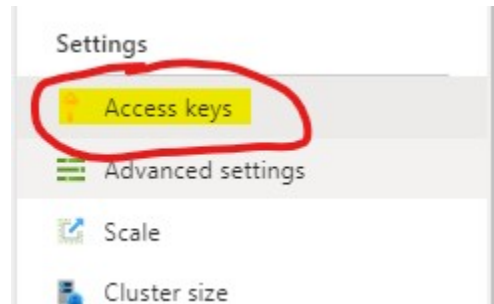
__3. In the search bar on the top of the page, search for **Azure Cache for Redis** and click on it.

Home >
Azure Cache for Redis 🔖 ...

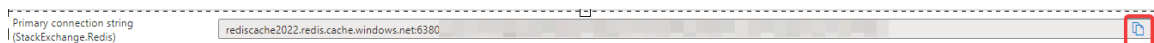
___ 4. Click the redis cache instance provided in the lab environment. For example:

<input type="checkbox"/> Name ↑↓	Location ↑↓	Status ↑↓
<input type="checkbox"/>  redisCache657	East US	Running

___ 5. In the navigation menu on the left hand side, click **Settings | Access Keys**.



___ 6. Click the copy button next to **Primary connection string**.



___ 7. Save the value in notepad since you will use it later in the lab.

Part 7 - Add Redis Cache support to the MVC application

In this part, you will add the Redis Cache support to the MVC application so it doesn't connect to the Azure SQL database each time the page is refreshed.

___ 1. In the terminal window, execute the following commands to install the Redis Cache and Newtonsoft.Json packages to the MVC application:

```
dotnet add package StackExchange.Redis --version 2.6.48
```

```
dotnet add package Newtonsoft.Json --version 13.0.1
```

__2. In VSCode, open **Controllers\HomeController.cs**.

__3. Below the existing using statements, add the following namespaces:

```
using StackExchange.Redis;  
using Newtonsoft.Json;
```

__4. In the **HomeController.cs** file, modify the existing *Users* method as shown in **bold**. [Empty out the code inside the Users method so it looks like this]:

```
public IActionResult Users()  
{  
}
```

__5. Inside the Users method, add the following code as shown in **bold**. (Note: Don't forget to use the actual redis cache connection string you obtained earlier in the lab):

```
public IActionResult Users()  
{  
    var context = new KafkaLabContext();  
  
    var redisConnectionString = "<Your Redis Cache Connection String>";  
  
    var redisCache =  
    ConnectionMultiplexer.Connect(redisConnectionString).GetDatabase();
```

__6. Below the above lines, add the following if-else block.

```
        if(string.IsNullOrEmpty(redisCache.StringGet("Users")))  
        {  
  
        }  
        else  
        {  
  
        }
```

7. Inside the if block, add the following lines of code as shown in **bold**, make sure to update the **<Your Student Id>** with your ID:

```
var message = "---Getting data from Azure SQL Database---";  
var users = context.UserRegistration<Your Student Id>.ToList();  
var serializedUsers = JsonConvert.SerializeObject(users);  
redisCache.SetString("Users", serializedUsers,  
    TimeSpan.FromMinutes(1));  
Console.WriteLine(message);  
ViewBag.Message = message;  
return View(users);
```

These lines obtain data from Azure Sql and cache them for one minute in Redis cache.

8. Inside the else block, add the following code, make sure to update the **<Your Student Id>** with your ID:

```
var message = "---Getting data from Redis Cache---";  
var cachedData =  
    JsonConvert.DeserializeObject<List<UserRegistration<Your Student  
    Id>>>(redisCache.StringGet("Users"));  
Console.WriteLine(message);  
ViewBag.Message = message;  
return View(cachedData);
```

These lines get called when data is already available in Redis cache.

Part 8 - Test the application with Redis Cache

In this part, you will test the application with Redis Cache. It will connect to the Azure SQL database if data isn't already available in Redis cache or if the data has expired in Redis cache.

__1. In the terminal window, execute the following command to start up the application:

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

__2. Open Chrome and navigate to the following URL:

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

__3. Click **View Users**.

Notice the user list shows up on the page. Also, notice the following message is displayed:

```
---Getting data from Azure SQL Database---
```

__4. Refresh the page in the browser.

Notice the following message is displayed on the page.

```
---Getting data from Redis Cache---
```

__5. Refresh the page a few more times and notice the data is retrieved from the cache.

__6. Click **Register**.

- ___7. Enter new values in to the fields and click the **Register User** button.
- ___8. Immediately switch back to the **View Users** page and notice the new record doesn't show up.
- ___9. Wait for a minute or so and refresh the **View Users** page and notice the new record shows up. You will also see the message that the records were retrieved from Azure SQL and then they will get cached again for another minute.

Part 9 - Clean-up

- ___1. In the VSCode terminal window, press Ctrl+C.
- ___2. Close VSCode.
- ___3. Close the browser window.

Part 10 - Review

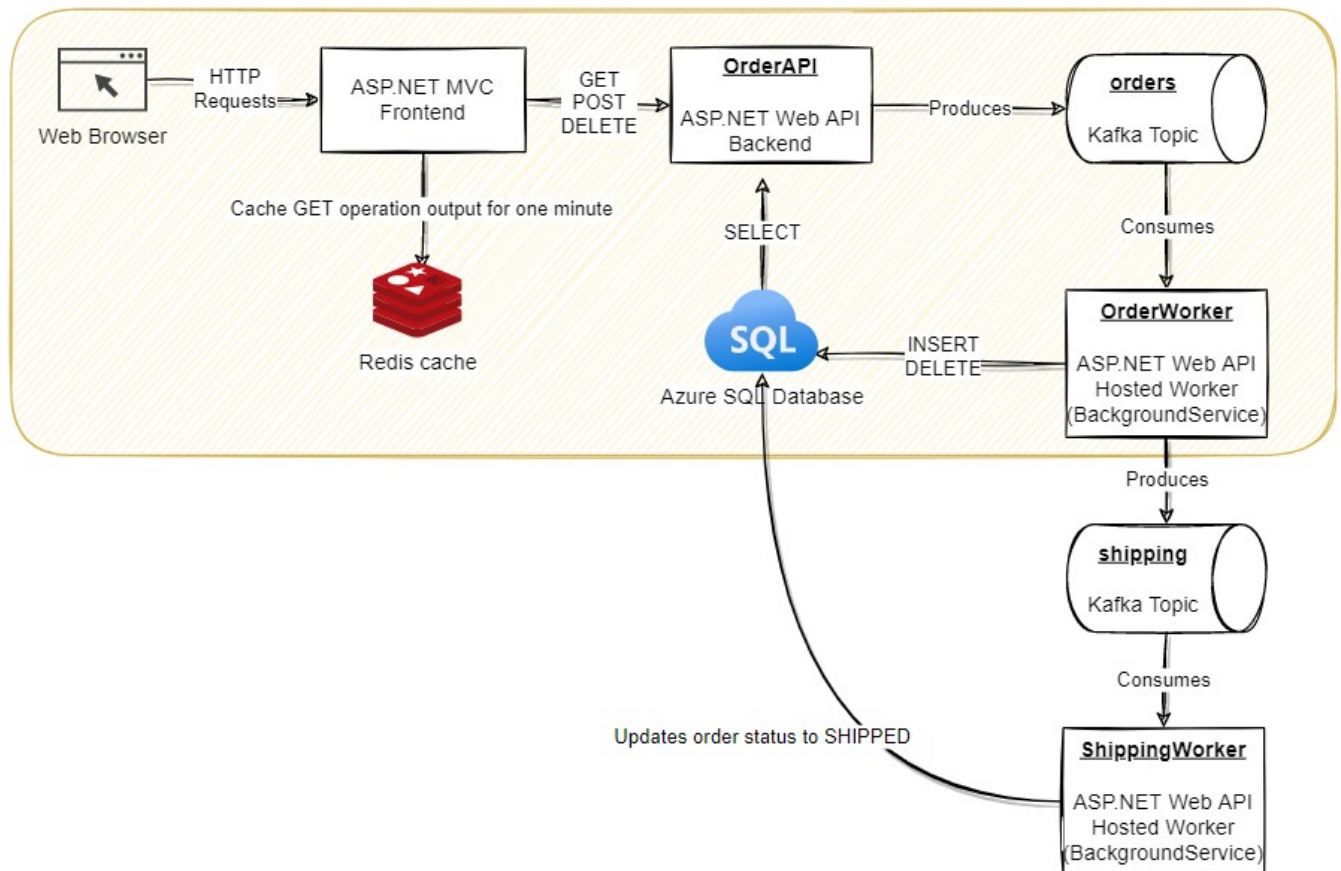
In this lab, you integrated Redis cache with .NET Core 3.1 ASP.NET MVC application.

Lab 11 - The Final Project

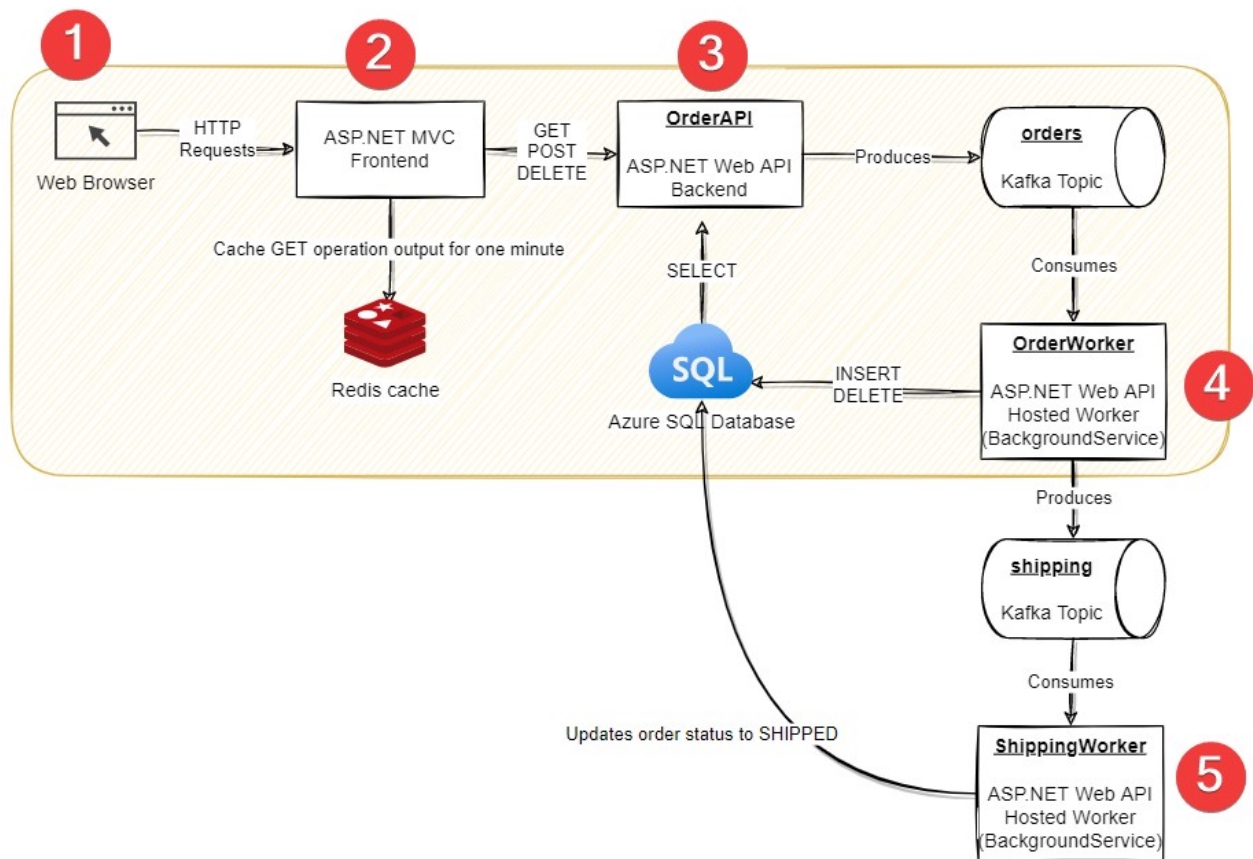
Now that we have seen various ways to integrate ASP.NET Core Web API, ASP.NET Core MVC, Kafka, Azure SQL, and Redis Cache, it is your turn to complete the Final Project by implementing the services to achieve the Kafka Microservice requirements in the next sections.

Part 1 - Overall Architecture

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

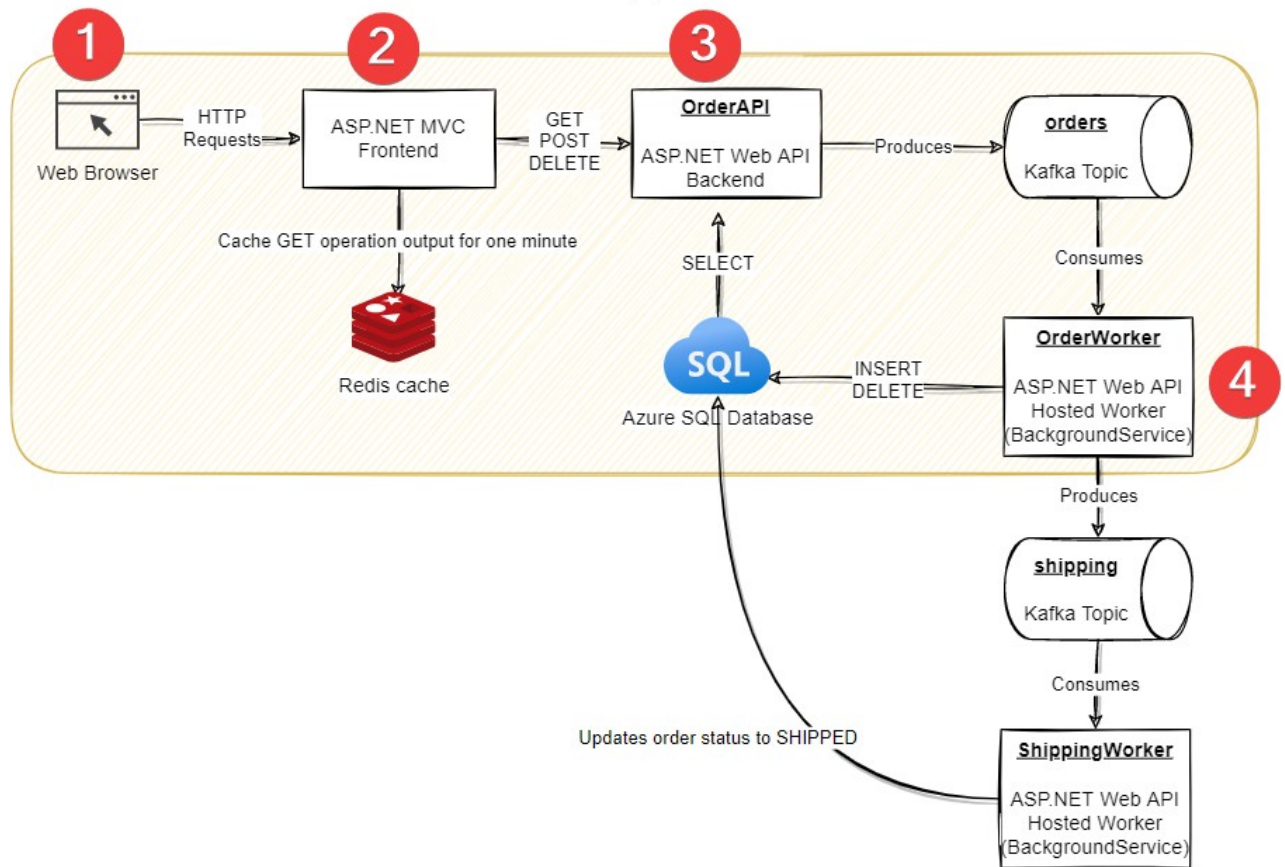


Part 2 - Order Placement - Workflow Diagram and Explanation



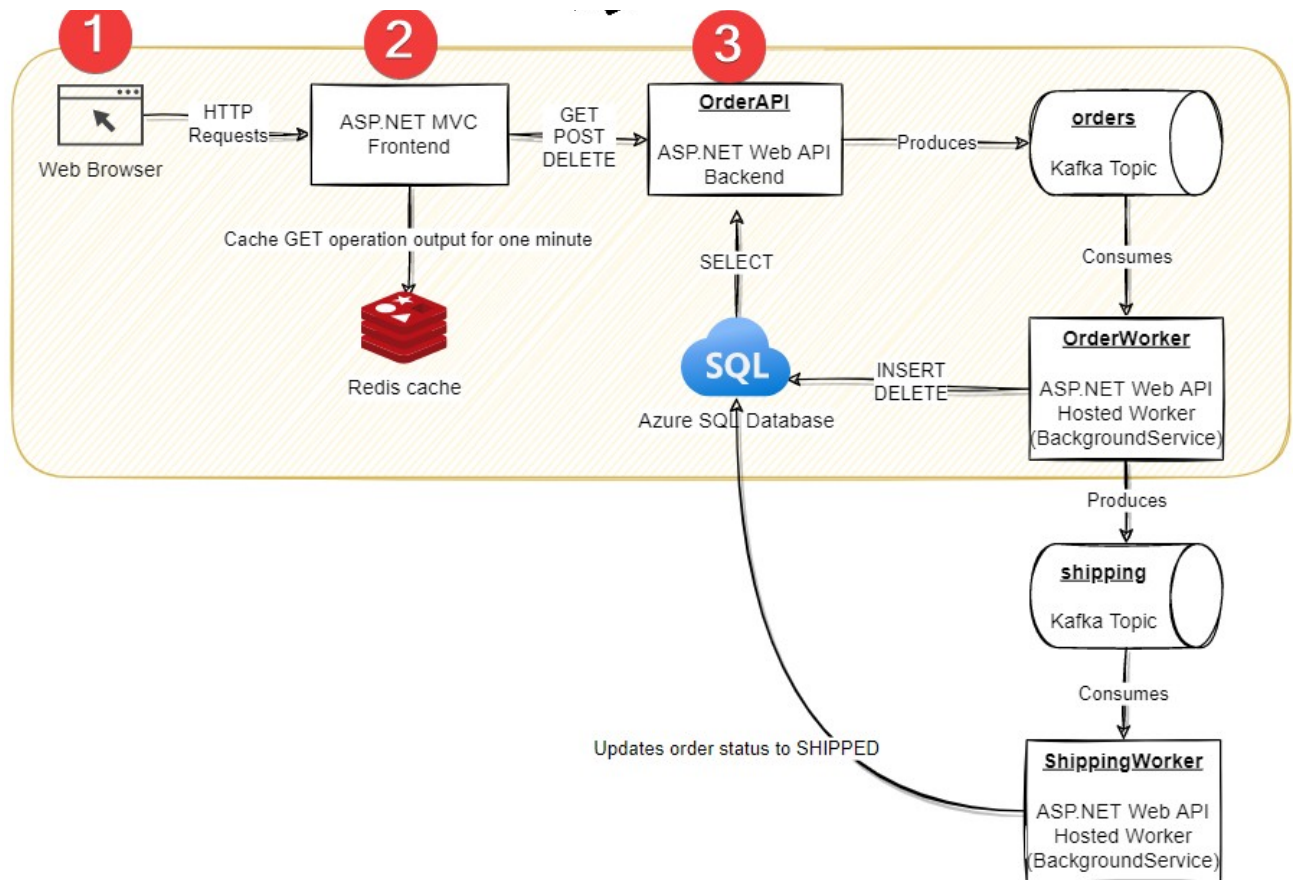
1. A user places an order request from a web browser.
2. The ASP.NET MVC frontend application receives the order request and calls the OrderAPI with the order information captured from the end-user.
3. OrderAPI receives the POST request from the MVC frontend and writes the order details to the orders 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 worker uses Entity Framework to connect to the Azure SQL database and inserts a record into the Order table. The order status is initially set to PLACED.
 1. **(OPTIONAL)**: OrderWorker can also act as a Kafka Producer and write order details to the shipping Kafka topic.
5. **(OPTIONAL)**: ShippingWorker is a Kafka consumer. The ShippingWorker worker/background service continuously runs and consumes messages/orders from the shipping Kafka topic. The worker uses Entity Framework to connect to Azure SQL and updates the order status from PLACED to SHIPPED.

Part 3 - 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.
3. OrderAPI queries the Azure SQL database and checks the following:
 1. If the status is PENDING, OrderAPI writes the order details to the orders Kafka topic.
 2. 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."
4. OrderWorker consumes messages/orders from the orders Kafka topic. The worker uses Entity Framework to connect to Azure SQL and delete 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 4 - Orders View - Workflow Diagram and Explanation



1. The end-user views the Order List page.
2. The MVC frontend does the following:
 1. Check if the order list is already cached in the Redis cache. If yes, return it to the browser without calling the OrderAPI backend. The cache should be configured to expire after one minute.
 2. If the order list isn't in the Redis cache, send 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 caches it in the Redis cache.

Part 5 - 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. The solution assumes the Order table in Azure SQL Database is Order00. Redis cache and Azure SQL database connection strings are removed from the solution. Kafka consumer and producer configuration is also 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. Azure SQL and Redis connection strings are currently hard-coded in C# in controller and Models\KafkaLabContext.cs files. If you are proficient in .Net, you can go ahead and configuration all connection strings 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.

→ ↻ ⓘ localhost:8080/Home/Order

kafka_project_mvc Home **Place Order** View Orders

Customer Name

Order Description

Order Amount

Place Order

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.

→ ↻ ⓘ localhost:8080/Home/View

kafka_project_mvc Home Place Order View Orders

Order Id	Customer Name	Order Description	Order Date	Order Amount	Order Status	
5a59bab5-5e2e-46d6-a6ef-97cb446ec9f9	Bob Smith	Mobile Phone	2022-01-02 10:10:00 AM	699.95	PENDING	DELETE
5a59bab5-5e2e-46d6-a6ef-97cb446ec9f8	Drew Smith	PS5	2022-02-03 5:06:00 AM	299.95	PLACED	DELETE
5a59bab5-5e2e-46d6-a6ef-97cb446ec9f7	Bob Smith	Laptop	2022-03-04 9:11:00 AM	1699.95	SHIPPED	DELETE

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 6 - Project Requirements

Here are the project requirements:

1. Order table

- a. Create a table named Order in the KafkaLab Azure SQL database. The table structure is described in the section titled Azure SQL Database - Order Table.

2. ASP.NET MVC Frontend

- a. Implement Redis cache support to the ASP.NET MVC frontend application. The data should be cached in the GET operation and should expire after one minute.
- b. Delete the cache on POST and DELETE operations by calling the Redis cache's KeyDelete("<key>") method.

3. OrderAPI

- a. Add Entity Framework support to the project.
- b. Implement the GET methods for GetOrders and GetOrderById to retrieve data from the Order table.
- c. Implement the POST method so it writes order details to the orders Kafka topic with the default status of PENDING.
- d. Implement the DELETE method. The method should use Entity Framework to connect to the Order table 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 use Entity Framework to 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 Entity Framework to change the order status to SHIPPED.

Part 7 - Azure SQL Database - Order Table

Use the same KafkaLab Azure SQL database that you used in the labs. Create a table named Order<Student Id>, e.g. Order01. (Note: All participants are sharing the same database and we don't want to have any clash. Therefore, you will create the Order table so it's unique. Throughout this project, we will refer to it as the Order table, but it is Order<Student Id>.)

The table should have the following structure.

1. OrderId: UNIQUEIDENTIFIER, PRIMARY KEY
2. CustomerName: VARCHAR(20), NOT NULL
3. OrderDescription: VARCHAR(20), NOT NULL
4. OrderDate: DATETIME, NOT NULL
5. OrderAmount: DECIMAL, NOT NULL
6. OrderStatus: VARCHAR(10), NOT NULL

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

Part 8 - 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 9 - Tips

1. To create a Web API project, use **dotnet new webapi --output my-project-name**.
2. Creating tables in a Azure SQL database is covered in the **Integrating Azure SQL and Kafka** lab.
3. 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.
4. 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**.
5. Entity Framework is covered in various labs, such as **Integrating Azure SQL and Kafka** and **Integrating Redis Cache with .NET Core 3.1 ASP.NET MVC**.
6. Redis cache is covered in the **Integrating Redis Cache with .NET Core 3.1 ASP.NET MVC** lab.
7. **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. Keep in mind that the GET operation will cache data for one minute and that can also delay the updated results from displaying in the frontend.**

