Hands On: Setup the Exercise Environment for Confluent Cloud and Python

Confluent Cloud Setup

Throughout this course, we'll introduce you to developing Apache Kafka event streaming apps with Python through hands-on exercises that will have you produce data to and consume data from Confluent Cloud. This exercise will get you set up for the exercises that follow.

Sign up for Confluent Cloud

Note: If you already have a Confluent Cloud account, you can skip this section and proceed to step 8.

If you haven't already signed up for Confluent Cloud, sign up now so when your first exercise asks you to log in, you are ready to do so.

1. Browse to the sign-up page: https://www.confluent.io/confluent-cloud/tryfree/
2. Fill in your contact information and a password. Be sure to remember these details as you will need them to log in to Confluent Cloud.
3. Click the **Start Free** button and wait for a verification email.
4. The link in the email will lead you to a screen where you will be prompted to complete a short survey. After doing so, click **Continue** or alternatively just click **Skip**.
5. When the information dialog appears that informs you of the promotional credit you can use to become familiar with Confluent Cloud, click **Get started** to continue with these set up steps.
6. When the next screen appears that prompts you to set up your first cluster, click **I'll do it later**.
7. Continue to step 9.

Login to the Confluent Cloud Console

8. Open URL https://confluent.cloud and log in to the Confluent Cloud console.
9. Navigate to the environments page and click **Add cloud environment**.
10. Name the new environment `learn-kafka-python` and click **Create**.

Enable Streams Governance

Next, you will be prompted to enable one of the available Streams Governance Packages. You need to do so since the course exercises utilize Schema Registry which these packages include.

11. Click **Begin configuration** for the **Essentials package**.
12. Select a provider and region and click **Enable**.

Create a Cluster in the `learn-kafka-python` Environment

13. Click **Create cluster on my own**.
14. Click **Begin configuration** for the **Basic** cluster.
15. Select a provider and region and click **Continue**.
16. Assign a name of `kafka-python` and click **Launch cluster**.

FREE ACCESS: Basic clusters used in the context of this exercise won't incur much cost, and the amount of free usage that you receive along with the promo code PYTHONKAFKA101 for $25 of free Confluent Cloud usage will be more than enough to cover it. You can also use the promo code CONFLUENTDEV1 to delay entering a credit card for 30 days.

Create and Download Python Client Configuration Properties

Next, you will create the configuration properties needed to connect to your cluster.

17. Navigate to the **Cluster Overview Dashboard** and for the **kafka-python** cluster.
18. Click on the **Python** tile in the **Connect to your systems** section.
19. Create a cluster API key and secret by clicking **Create Kafka cluster API key**.
20. Review the key and the secret in the pop-up and click **Download and continue**.

    This will download the key and secret to the local machine **Downloads** directory.
21. To create the Schema Registry API key and secret, repeat the process by clicking **Create Schema Registry API key**.
22. Review the sample Python client configuration settings. The API keys and secrets should be populated in the properties in the middle of the page. If they are not visible, select the **Show API keys** option.
23. Click the **Copy** button.

Setup Project

Create Python Dictionary

24. Create a file named `python.properties` located in the `~/.confluent` directory.

    Note: The `.confluent` directory is the default location for Confluent Platform related configuration files.
25. Edit `python.properties` and paste the previously copied Python client Confluent Cloud properties.
26. Save `python.properties` but leave it open in your IDE.
27. Create a config directory named `kafka-python` for the course exercises and `cd` into it.
28. Create a file named `config.py` that will contain a Python dictionary.
29. Add the following lines to `config.py`:

    ```
    config = {

         'bootstrap.servers': '<bootstrap-server-endpoint>',

         'security.protocol': 'SASL_SSL',

         'sasl.mechanisms': 'PLAIN',

         'sasl.username': '<CLUSTER_API_KEY>',

         'sasl.password': '<CLUSTER_API_SECRET>'}
    ```
30. Update the values for `bootstrap.servers`, `sasl.username`, and `sasl.password` to the values contained in `python.properties`.

    We'll add more to `config.py` in later exercises.

Install Python 3.X and Related Prerequisites on Local Machine

The course exercises depend upon Python 3.X and several related packages being installed. The steps in this prerequisites section satisfy this dependency. Complete the steps as needed on your local machine.

Note: These steps correspond to installing these packages on Ubuntu 20.04. Adjust the steps as needed to suit your local environment.

31. First update all the packages in your local machine:

    ```
    sudo apt update
    ```

32. If any of the system packages on your local machine need to be upgraded, run the following command to do so:

```
sudo apt upgrade
```

33. Install `python3`:

```
sudo apt install python3
```

34. Install `pip3`:

```
sudo apt install python3-pip
```

35. Identify default `python` version:

```
python --version
```

The version shown in the command response should be similar to `Python 3.X.X`. If a non-Python 3.X version is returned, you will need to either update your alias for the `python` command or explicitly run the `python3` and `pip3` commands during the exercises for this course.

36. Install `virtualenv`:

```
pip install virtualenv
```

Create a New Virtual Environment with `virtualenv`

37. Create a virtual environment:

```
virtualenv kafka-env
```

38. Activate the virtual environment:

```
source kafka-env/bin/activate
```

Install Confluent Kafka

39. Run the following command:

```
pip install confluent-kafka
```

You're now ready for the upcoming exercises.

Hands On: Use the Python Producer Class

Use Producer to Send Events to Kafka

In this exercise, you will use the `Producer` class to write events to a Kafka topic.

Create Topics

1. In the Confluent Cloud Console, navigate to the **Topics** page for the **kafka-python** cluster in the **learn-kafka-python** environment.
2. Create a new topic called `hello_topic`, with `6` partitions and defaults settings.

Project Setup

3. Open a terminal window and navigate to the `kafka-python` directory that you created in the previous exercise.
4. If you are not currently using the `kafka-env` environment that was created in the last exercise, switch to it with the following command:

```
source kafka-env/bin/activate
```

5. Create a file called `producer.py`.
6. Open `producer.py` in an IDE of your choice.

Add Required Imports

7. Add the following import statements to the top of the `producer.py` file:

```
from confluent_kafka import Producer

from config import config
```

Create callback function

8. Create a function called `callback()` that can be passed to the produce() method.

```
def callback(err, event):

    if err:
        print(f'Produce to topic {event.topic()} failed for event:
{event.key()}')
    else:
        val = event.value().decode('utf8')
        print(f'{val} sent to partition {event.partition()}.')
```

Create function to produce to hello_topic

9. Create a function called `say_hello()` that takes a producer and a key.

```
def say_hello(producer, key):
    value = f'Hello {key}!'
    producer.produce('hello_topic', value, key, on_delivery=callback)
```

Add Main Block

10. Add the following main block that will pull this all together

```
if __name__ == '__main__':
    producer = Producer(config)
    keys = ['Amy', 'Brenda', 'Cindy', 'Derrick', 'Elaine', 'Fred']
    [say_hello(producer, key) for key in keys]
    producer.flush()
```

Run the Program

11. Execute the program by running the following command

```
python producer.py
```

Notice how the different names, which we are using for keys, result in specific partition assignments. To get a better idea of how this works, you can try changing some of the names and see how the partition assignment changes.

Hands on: Use the Python Consumer Class

Use Consumer to Read Events from Kafka

In this exercise, you will use the Consumer class to read events from a Kafka topic.

Topics

We will be using the `hello_topic` which we created in the **Use Producer to Send Events to Kafka** exercise. We will be reading these events during this exercise so if you did not complete the previous exercise, you will need to either do so or you can use the Confluent Cloud Console to manually create the topic and send some events to it.

Project Setup

1. Open a terminal window and navigate to the `kafka-python` directory that you created in the previous exercise.
2. If you are not currently using the `kafka-env` environment that was created in the last exercise, switch to it with the following command:

   ```
   source kafka-env/bin/activate
   ```

3. Create a file called `consumer.py`.
4. Open `consumer.py` in an IDE of your choice.

Add Required Imports

5. Add the following import statements to the top of the `consumer.py` file:

   ```
   from confluent_kafka import Consumer, KafkaException
   from config import config
   ```

Create function to update configuration

6. Create a function called `set_consumer_configs()`. We will call this method in the main block in order to add some consumer specific configuration properties.

```
def set_consumer_configs():

    config['group.id'] = 'hello_group'

    config['auto.offset.reset'] = 'earliest'

    config['enable.auto.commit'] = False
```

Along with setting the `group.id`, we are also setting `auto.offset.reset` to `earliest` so that we can consume events that are already in our target topic. Finally, we are setting `enable.auto.commit` to `False` so that we can control the committing of offsets for our consumer.

Create callback function for partition assignment

7. Create a function called `assignment_callback()` that takes a consumer and a key.

```
def assignment_callback(consumer, partitions):

    for p in partitions:

        print(f'Assigned to {p.topic}, partition {p.partition}')
```

This callback will be passed into the `consumer.subscribe()` call and it will be invoked whenever topic partitions are assigned to this consumer. This includes during the `subscribe()` call, as well as any subsequent rebalancing.

Add Main Block

8. Add the following beginnings of the main block.

```
if __name__ == '__main__':
```

```
set_consumer_configs()

consumer = Consumer(config)

consumer.subscribe(['hello_topic'], on_assign=assignment_callback)
```

Here we are updating the `config` Dictionary, creating our `Consumer` and then calling the `subscribe()` method. This method takes a list of topic names. This is often a list of one. We are passing a function for the `on_assign` callback. We could also pass in functions for `on_revoke` and `on_lost`, but these are less commonly used.

9. Add a `try|except|finally` block.

```
try:

except KeyboardInterrupt:

    print('Canceled by user.')

finally:

    consumer.close()
```

We're going to be adding an endless loop in the next step, so we'll use the `except KeyboardInterrupt` to catch a CTRL-C to stop the program, and then we'll call `consumer.close()` in the finally block to make sure we clean up after ourselves.

10. To add the poll loop, add the following `while` loop between the `try:` and `except`.

```
    while True:

        event = consumer.poll(1.0)

        if event is None:

            continue

        if event.error():

            raise KafkaException(event.error())

        else:

            val = event.value().decode('utf8')

            partition = event.partition()

            print(f'Received: {val} from partition {partition}    ')

            # consumer.commit(event)
```

In our loop, we call `consumer.poll()` repeatedly and then print the value of the event that is received, along with the partition that it came from. Notice that the `consumer.commit()` call is commented for now.

Run the Consumer

11. Execute the program by running the following command

    ```
    python consumer.py
    ```

    If you ran the producer exercise earlier, you should see output like this:

    ```
    Assigned to hello_topic, partition 0

    Assigned to hello_topic, partition 1

    Assigned to hello_topic, partition 2

    Assigned to hello_topic, partition 3

    Assigned to hello_topic, partition 4

    Assigned to hello_topic, partition 5

    Received: Hello Fred! from partition 2

    Received: Hello Cindy! from partition 0

    Received: Hello Amy! from partition 1

    Received: Hello Elaine! from partition 5

    Received: Hello Derrick! from partition 4
    Received: Hello Brenda! from partition 3
    ```

    The first six lines are the partition assignments, which are all done in order. The next six are the events that are in our `hello_topic` Let's run the consumer again and observe the result.
12. To stop the consumer, press **Ctrl+C**.
13. Run the consumer again:

    ```
    python consumer.py
    ```

    Notice we get the same result. This is because we have `enable.auto.commit` set to `False`, `auto.offset.reset` set to `earliest`, and we are not committing our offsets.

14. To stop the consumer, press **Ctrl+C**.
15. Uncomment the `consumer.commit(event)` call.
16. Run the consumer, stop it by pressing **Ctrl+C**, and run the consumer again.

    You should still see the same result from the first run but notice subsequent run(s) just show the assignment lines and wait for more events.

Produce New Events

Let's leave the consumer running and confirm we see the new events when they are produced to `hello-topic`.

Note: This section depends upon the producer that was created in the **Use Producer to Send Events to Kafka** exercise.

17. Open a new terminal and run the producer:

```
source kafka-env/bin/activate && \
python producer.py
```

    Return to the terminal the consumer is running in where you will see the newly produced events appear.

Observe Consumer Group Rebalance Behavior

Let's see what happens when we add a consumer instance to the 'hello_group' consumer group.

18. Open a new terminal and run the consumer:

```
source kafka-env/bin/activate && \
python consumer.py
```

    Notice how you only see three lines of partition assignments. If you go back to the terminal window that already had the consumer running, you will see the other three partition assignments. Starting the second instance of our consumer caused it to be added to the consumer group with the existing instance, triggering a rebalance. If you launch a third instance, they will each have two partition assignments.

Now let's see how the Kafka Consumer Group Protocol handles a failed consumer instance.

19. Stop the second consumer instance.
20. Return to the terminal window where the first consumer instance is running.

    Notice that a fresh set of partition assignments appear for all six partitions. When the other consumer instance was stopped it removed itself from the group triggering a consumer group rebalance. This rebalance would have also occurred if the consumer instance had failed.