Airflow & Kafka 연동 가이드 v2

Table of Contents

---

1. 문서 개요
    1. 목적
    2. 범위
    3. 시스템 구성도
    4. 참고자료
    5. 버전
2. Prerequisite
    1. Airflow VM 구축
    2. Docker 및 Docker Compose 설치 (*Kafka VM only*)
3. Guide
    1. Kafka 설치
    1. Docker Compose 파일 작성하기
    2. Kafka 실행하기
    2. Airflow Kafka 연동
    1. 통신 체크
    2. Airflow UI에서 Connection 설정 하기
    3. Airflow-Kafka 연동 테스트

1. 문서 개요

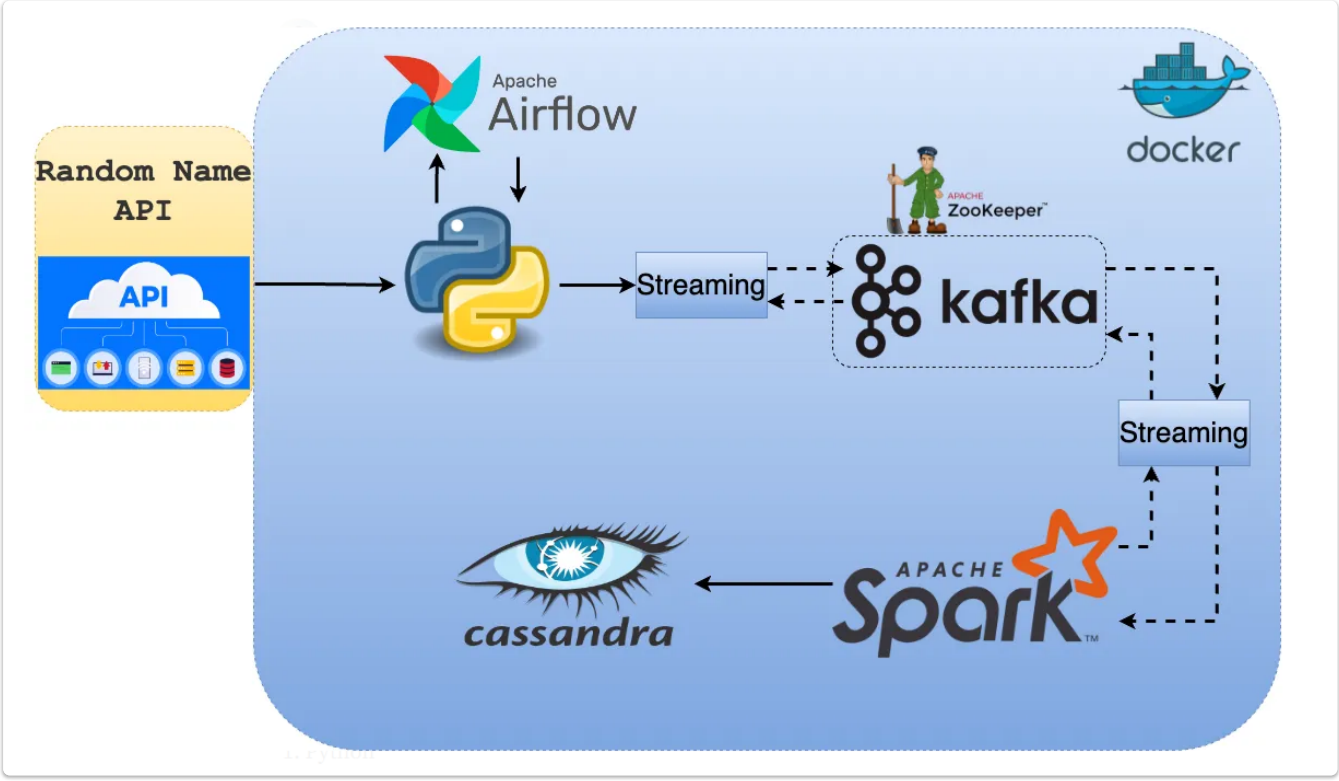
1.1 목적

본 문서는 AirFlow와 Kafka를 연동하여 실시간으로 데이터를 처리하고 작업을 스케줄링 및 모니터링 하는 방법을 기술 하였다.

1.2 범위

설치 범위는 Airflow와 Kafka의 연동을 기준으로 작성 하였다.

1.3 시스템 구성도



1.4 참고자료

> *Kafka install and run*

1.5 버전

Docker	**2.24**
Docker Compose	2.24.5
confluentinc	6.1.15
zookeeper	3.8.3
kafka	2.7.x
java	1.8, 11

2. Prerequisite

본 설치 가이드는 Ubuntu 22.04 환경에서 설치하는 것을 기준으로 작성하였다. Airflow, Kafka 각각의 VM을 구축하는 것을 기본으로 총 두 개의 VM이 필요하다.

2.1 Airflow VM 구축

2.2 Docker 및 Docker Compose 설치 (*Kafka VM only*)

3. Guide

3.1 Kafka 설치

3.1.1 Docker Compose 파일 작성

Docker Compose 파일을 작성하고 컨테이너를 생성하여 Kafka를 설치한다.

- kafka 설치를 위해 작업 디렉토리를 생성하고 이동한다.

```
mkdir $HOME/kafka && cd $HOME/kafka
```

- docker-compose 파일을 생성한다.

```
vi docker-compose.yaml
```

```
version: '2'

networks:
  kafka_default:
    driver: bridge

services:
  zookeeper:
    image: confluentinc/cp-zookeeper:6.1.15
    hostname: zookeeper
    container_name: zookeeper
    ports:
      - "2181:2181"
    environment:
      ZOOKEEPER_CLIENT_PORT: 2181

  broker:
    image: confluentinc/cp-kafka:6.1.15
    hostname: broker
    container_name: broker
    depends_on:
      - zookeeper
    ports:
      - "19092:19092"
      - "9092:9092"
    environment:
      KAFKA_BROKER_ID: 1
      KAFKA_ZOOKEEPER_CONNECT: 'zookeeper:2181'
      KAFKA_LISTENER_SECURITY_PROTOCOL_MAP: 'INTERNAL:PLAINTEXT,EXTERNAL:PLAINTEXT'
      KAFKA_ADVERTISED_LISTENERS: 'INTERNAL://broker:9092,EXTERNAL://133.186.240.216:19092'
      KAFKA_INTER_BROKER_LISTENER_NAME: INTERNAL
      KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 1
      KAFKA_TRANSACTION_STATE_LOG_MIN_ISR: 1
      KAFKA_TRANSACTION_STATE_LOG_REPLICATION_FACTOR: 1
      KAFKA_GROUP_INITIAL_REBALANCE_DELAY_MS: 0
```

Note

주요 환경 변수 값에 대한 설명

- `network` : `docker network ls` 명령어를 통해 생성된 `bridge` 네트워크를 입력한다.
- `broker.ports` : Kafka 내부에서는 `9092` port를 사용하고 외부에서는 `19092` 를 사용한다.
- `broker.enviroment`
  - `KAFKA_BROKER_ID` : Airflow에서 broker를 구분하기 위한 id이다.
  - `KAFKA_LISTENER_SECURITY_PROTOCOL_MAP` : 내부와 외부를 구분하는 protocol 명을 입력한다.
  - `KAFKA_ADVERTISED_LISTENERS` : 외부에서 접근하기 위해서 꼭 필요한 변수이며, 외부에서 접근하는 url는 *vm의 public ip*와 외부에서 사용하는 *19092* 포트를 명시한다.

- Docker Compose 파일을 실행한다.

```
docker compose up -d
```

- 설치가 정상적으로 되었는지 확인한다.

```
docker ps
```

```
CONTAINER ID   IMAGE                              COMMAND                  CREATED        STATUS
PORTS
NAMES
1796d96f4eac   confluentinc/cp-kafka:6.1.15       "/etc/confluent/dock…"   43 hours ago   Up 43 hours
0.0.0.0:9092->9092/tcp, :::9092->9092/tcp, 0.0.0.0:9101->9101/tcp, :::9101->9101/tcp, 0.0.0.0:19092-
>19092/tcp, :::19092->19092/tcp   broker
b98d84e1fe86   confluentinc/cp-zookeeper:6.1.15   "/etc/confluent/dock…"   43 hours ago   Up 43 hours
2888/tcp, 0.0.0.0:2181->2181/tcp, :::2181->2181/tcp, 3888/tcp
zookeeper
```

3.1.2 Kafka 실행하기

Kafka의 동작 원리를 이해하기 위해 토픽을 생성하여 Producer, Consumer를 실행 한다.

- Topic을 생성한다.

```
docker compose exec broker kafka-topics --create --topic my-topic --bootstrap-server broker:9092 --
replication-factor 1 --partitions 1
```

> `--topic`: *사용할 토픽의 이름*
> `--bootstrap-server`: *Listener의 타입에 따른 서버 주소*

- 생성된 topic을 확인한다

```
docker compose exec broker kafka-topics --describe --topic my-topic --bootstrap-server broker:9092
```

- Producer, Consumer를 실행한다.

> *실시간으로 데이터의 송수신을 확인 하기 위해 Consumer, Producer 두 개의 창을 띄운다.*

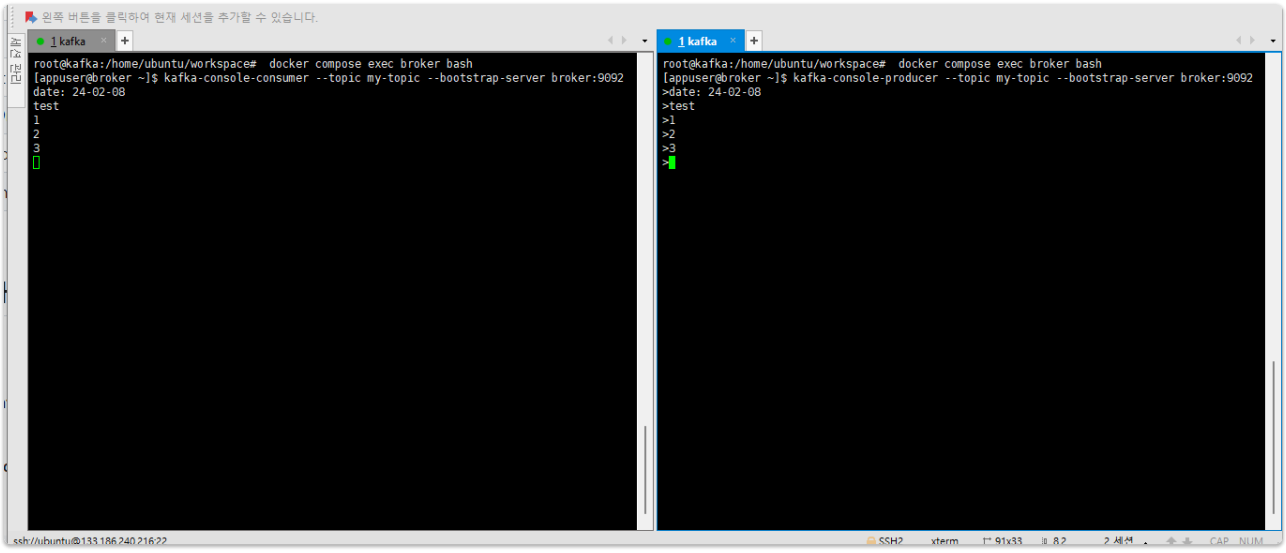
- consumer를 실행한다.

```
$ docker compose exec broker bash
.
.
[appuser@94e94072e1ea ~]$ kafka-console-consumer --topic my-topic --bootstrap-server broker:9092
```

- producer를 실행한다.

```
$ docker compose exec broker bash
.
.
.
[appuser@94e94072e1ea ~]$ kafka-console-producer --topic my-topic --bootstrap-server broker:9092
```

- 메세지가 실시간으로 송수신이 된다면 kafka가 정상적으로 동작 하고 있는 것이다.



3.2 Airflow Kafka 연동

Airflow와 Kafka를 연동하기 위해 사전 준비를 한다.

3.2.1 통신 체크

Kafka의 Container와 Airflow의 웹 서버의 통신을 체크한다.

```
$ nc -vz {VM_MASTER_IP} {OPEN_PORT} #내가 확인하고 싶은 포트
```

- Airflow vm에서 Kafka Broker 의 외부 포트로 연결을 확인한다.

```
nc -vz 133.186.240.216 19092
Connection to 133.186.240.216 19092 port [tcp/*] succeeded!
```

- Kafka Broker Container 내부에서 Airflow Server로 연결을 확인한다.

```
nc -vz 133.186.155.37 8080
Ncat: Version 7.92 ( https://nmap.org/ncat )
Ncat: Connected to 133.186.155.37:8080.
Ncat: 0 bytes sent, 0 bytes received in 0.01 seconds.
```

- Kafka Broker가 선언한 `KAFKA_ADVERTISED_LISTENERS`의 동작을 확인한다.

> *bootstrap-server를 외,내부로 선언하여 topic list를 확인한다.*

```
[appuser@broker ~]$ kafka-topics --list --bootstrap-server 133.186.240.216:19092
__consumer_offsets
my-topic
---
[appuser@broker ~]$ kafka-topics --list --bootstrap-server broker:9092
__consumer_offsets
my-topic
```

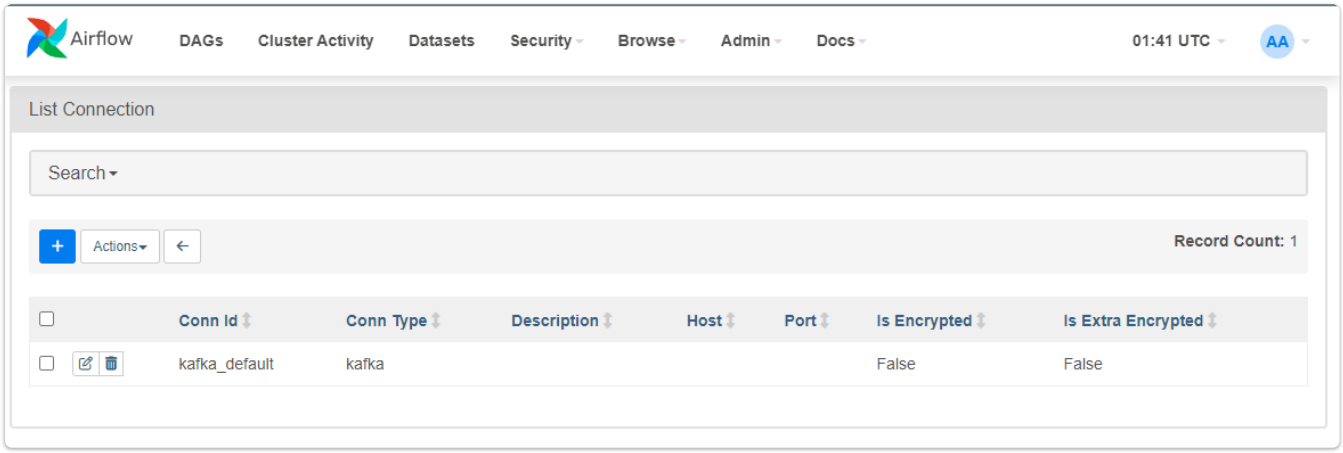
3.2.2 Airflow UI에서 Connection 설정 하기

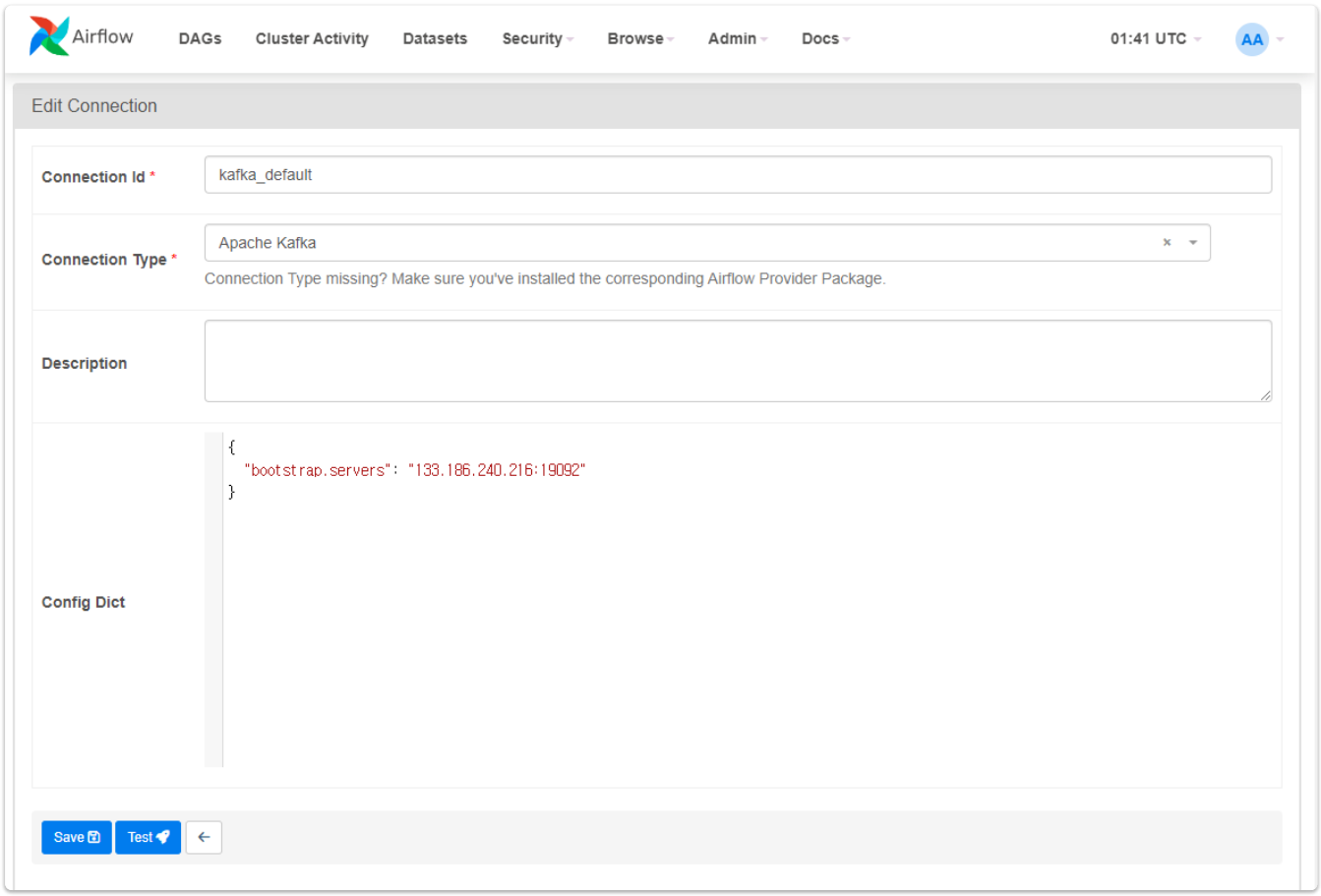
Connections에는 Connections Type이 존재 하는데 `conn type`에 `Apache Kafka`가 없다면 아래의 설치를 진행한다.

❗ *Airflow의 작업 디렉토리에서 진행해야 하며 vertiualenv를 실행 해준다.*

```
pip install airflow-provider-kafka
pip install apache-airflow-providers-apache-kafka==1.3.1
pip install confluent-kafka
pip install kafka-python
```

- Airflow UI에서 `Admin > Connections`에서 `kafka_default`를 수정한다.
  - `connection type`은 `Apache Kafka`로 변경한다.
  - `bootstrapt.servers`는 Kafka Broker의 `EXTERNAL` 주소로 변경한다.





3.3 Airflow-Kafka 연동 테스트

python 코드로 작성된 Dag 파일을 통해 Kafka와 정상적으로 연동이 되었는지 테스트 한다.

테스트 시나리오
네트워크 연결 체크
토픽 생성
메세지 송신

테스트 시나리오
메세지 수신
테스크 완료

- Python 코드를 airflow 작업 디렉토리의 `/dags` 에 생성 후 저장한다.

> *`kafkatest.py` 의 `bootstrap_servers`, `topic_name` 변수값은 알맞게수정한다.*

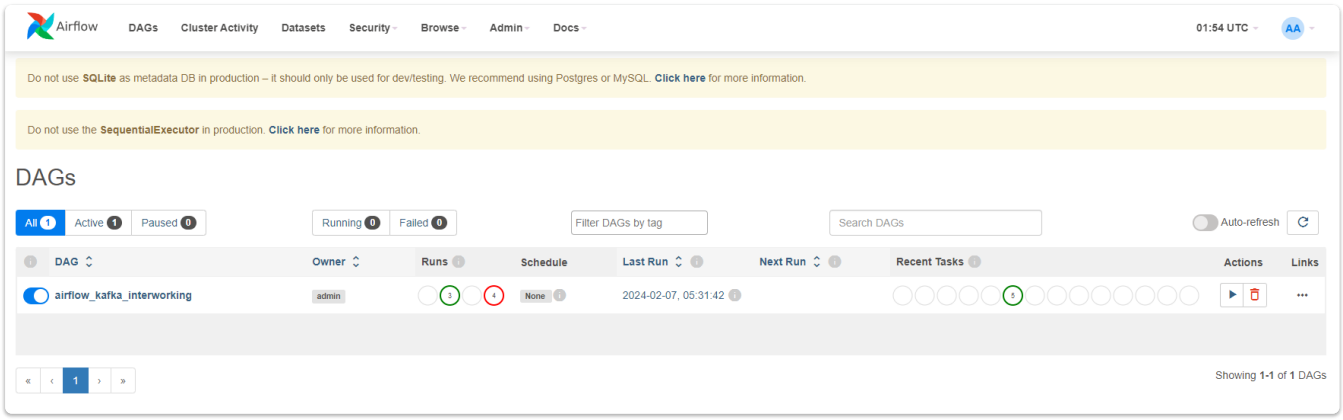
```
cd $HOME/airflow/dags
vi kafkatest.py
```

```
from airflow import DAG
from airflow.operators.python_operator import PythonOperator
from kafka import KafkaProducer, KafkaConsumer
from kafka.admin import KafkaAdminClient, NewTopic
from datetime import datetime
from airflow.models import TaskInstance
from kafka.errors import TopicAlreadyExistsError

# Define Kafka broker
bootstrap_servers = '133.186.240.216:19092'
topic_name = 'airflow-topic'

# Function to check if Kafka is connected
def check_connected():
    try:
        producer = KafkaProducer(bootstrap_servers=bootstrap_servers)
        consumer = KafkaConsumer(topic_name, bootstrap_servers=bootstrap_servers)
        return True
    except Exception as e:
        print(f"Error connecting to Kafka: {str(e)}")
        return False

# Function to create Kafka topic
def create_topic():
    admin_client = KafkaAdminClient(bootstrap_servers=bootstrap_servers)
    existing_topics = admin_client.list_topics()
    # 토픽이 이미 생성 되어 있다면 생성 되어 있는 토픽을 가져온다.
    if topic_name in existing_topics:
        print(f"Topic '{topic_name}' already exists.")
    else:
        topic = NewTopic(name=topic_name, num_partitions=1, replication_factor=1)
        try:
            admin_client.create_topics([topic])
            print(f"Topic '{topic_name}' created successfully.")
        except TopicAlreadyExistsError:
            print(f"Topic '{topic_name}' already exists.")


# Function to produce a message to Kafka
def produce_message():
    producer = KafkaProducer(bootstrap_servers=bootstrap_servers,linger_ms=20)#실시간 스트리밍이 아닌 20초 간격으로 메세지를 전송한다.
    message = f"Hello from Airflow. Message produced at {datetime.now()}"
    producer.send(topic_name, message.encode())
    producer.flush()

# Function to consume a message from Kafka
def consume_message(**context):
    consumer = KafkaConsumer(topic_name, bootstrap_servers=bootstrap_servers, auto_offset_reset='earliest')
    try:
```

```
        message = next(consumer)
        print(f"Message consumed: {message.value.decode()}")
        # Mark the task as success
        context['task_instance'].log.info("Message consumed successfully")
        context['task_instance'].state = "success"
    except StopIteration:
        # No message available, mark the task as failed
        context['task_instance'].log.info("No message available to consume")
        context['task_instance'].state = "failed"
    finally:
        # Close the consumer to release resources
        consumer.close()

# Function to finish
def finish():
    print("All tasks completed.")

# Define the DAG
default_args = {
    'owner': 'admin',
    'depends_on_past': False,
    'start_date': datetime(2024, 2, 7),
    'retries': 1,
}

dag = DAG(
    'airflow_kafka_interworking',
    default_args=default_args,
    description='DAG for testing interworking of Airflow and Kafka',
    schedule_interval=None,
)

# Define tasks
create_topic_task = PythonOperator(
    task_id='create_topic',
    python_callable=create_topic,
    dag=dag,
)

check_connected_task = PythonOperator(
    task_id='check_connected',
    python_callable=check_connected,
    dag=dag,
)

produce_message_task = PythonOperator(
    task_id='produce_message',
    python_callable=produce_message,
    dag=dag,
)

consume_message_task = PythonOperator(
    task_id='consume_message',
    python_callable=consume_message,
    dag=dag,
)

finish_task = PythonOperator(
    task_id='finish',
    python_callable=finish,
    dag=dag,
)

# Define task dependencies
```

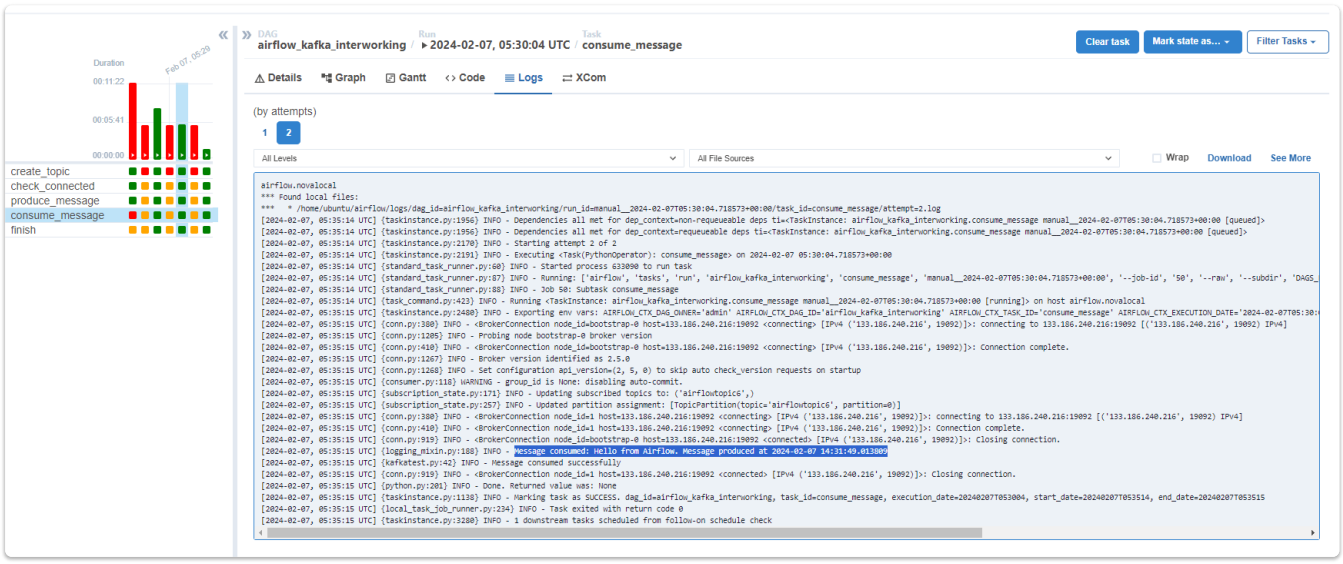
```
check_connected_task >> create_topic_task  >> produce_message_task >> consume_message_task >> finish_task
```

- Airflow의 UI에 접속하여 생성된 Dag를 확인하고 실행한다.



- 로그를 확인하며 각 Function의 상태를 확인한다.

> *Consume_message의 Log에 produce_message를 수행하며 생성된 Message가 출력된다.*



Airflow와 Kafka가 서로 메시지를 주고받고 정상적으로 작동하고 있으므로 연결이 완료되었음을 확인할 수 있다.