Airflow

Airflow란?

Airflow = Scheduler

Airflow = Orchestrator

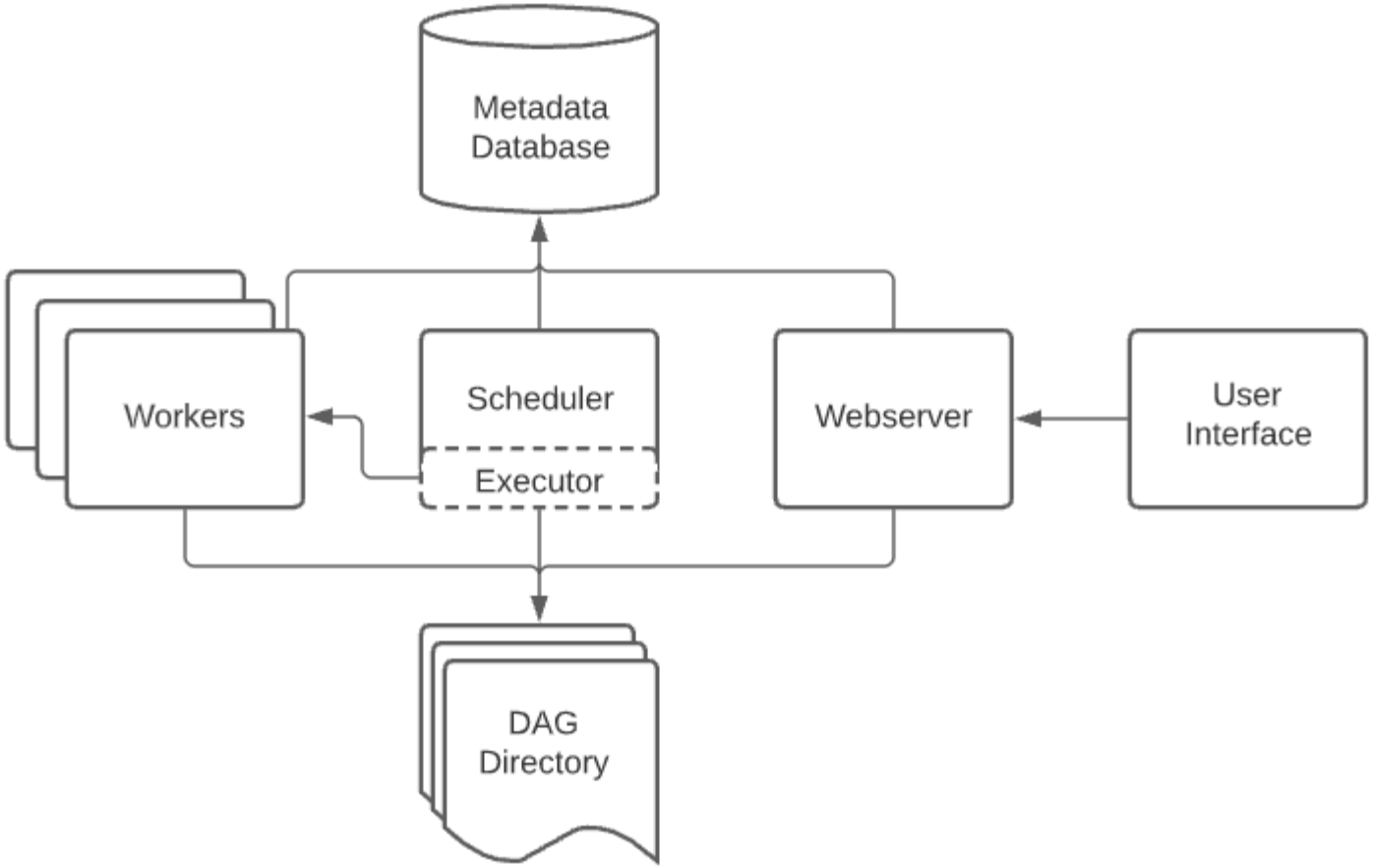
Airflow의 장점?

- Python 코드로 쉽고 간단하게 다양한 batch 작업, workflow를 구성할 수 있다.
- Python 코드로 dynamic한 pipeline을 구성할 수 있다.
- 여러 사람이 동시에 workflow를 개발하고, 개별 workflow 별로 동작과 설정 등을 관리할 수 있다.
- Operator로 반복 작업을 줄이고, 다양한 기술 스택과 연결할 수 있다.
- Jinja template으로 parameterize를 할 수 있다.
- Batch를 처리하는 인프라를 동적으로 늘리거나 줄일 수 있다.
- Batch처리 결과에 대한 수정을 동적으로 할 수 있다.
- 직관적이고 편리한 Web UI 관리 툴로 다양한 배치 작업을 쉽게 시각화하거나 디버깅 할 수 있다.

Airflow의 한계

- 스트리밍 작업
- Airflow 외부 요소에 의해 trigger 되는 scheduling 방식
- 지연을 허용하지 않는 작업의 스케줄링
- Airflow worker 내부에서의 고부하 작업

Airflow의 아키텍처?



❖ Scheduler
- 스케줄된 workflow를 trigger하고 task를 executor에게 실행하도록 제출하는 역할을 한다.
- 스케줄러는 모든 task와 DAG를 모니터링하고, 각 task instance를 조건에 맞게 trigger한다.
- (default) 1분에 한번씩 모든 DAG들의 정보를 업데이트하고, 상태정보를 확인하고 스케줄링 한다.

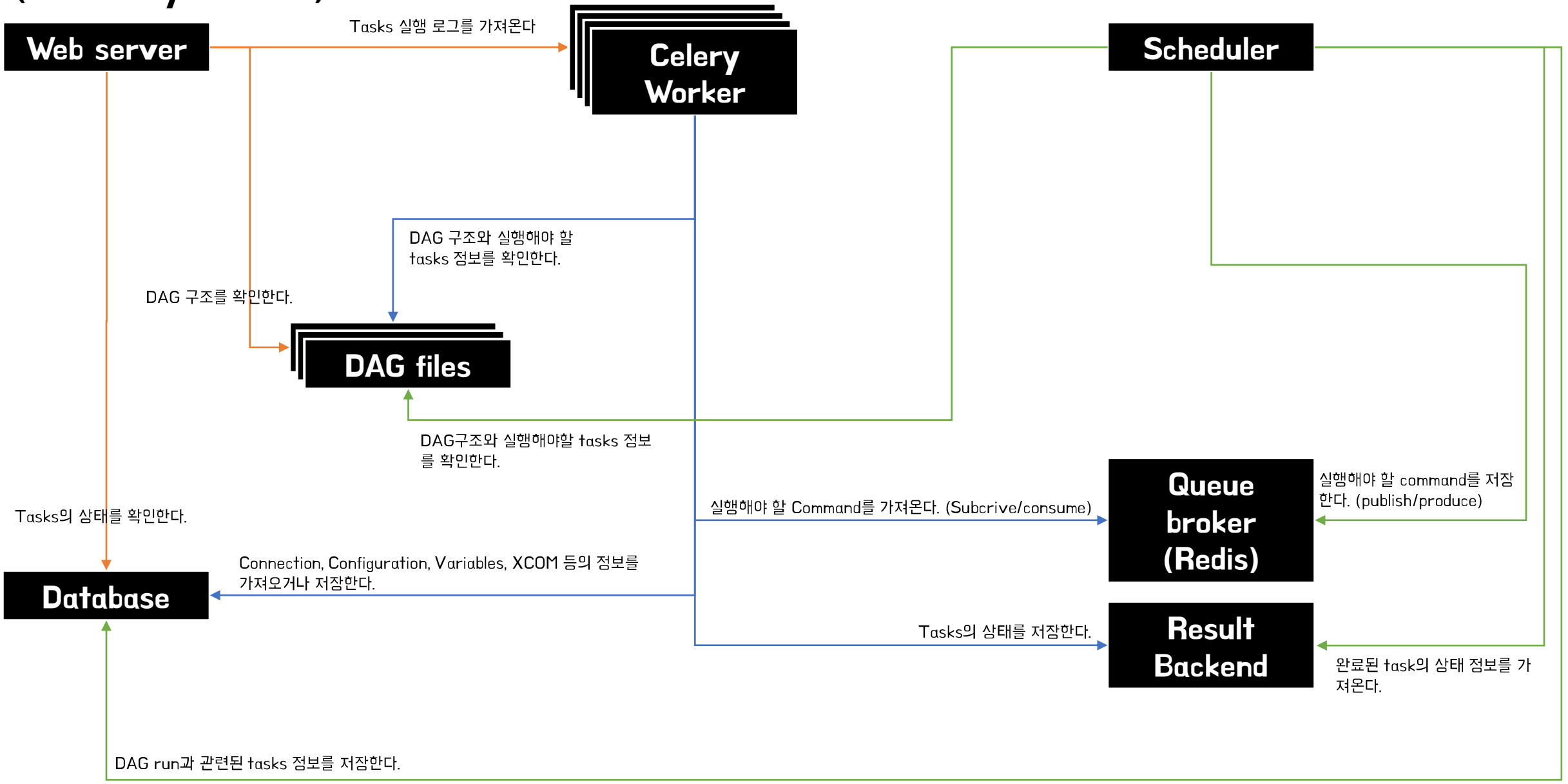
❖ Executor
- Task의 실행을 관리한다.
- 실행은 스케줄러 내부에서 수행할 수도 있지만 일반적으로 외부의 worker에게 맡긴다.

❖ Webserver
- 유저가 DAG나 task의 상태를 관리하는 web interface 도구이다.
- 여러 공통설정을 관리하거나, 매뉴얼한 작업의 수정, 디버깅까지 가능하다.

❖ DAG Directory
- Workflow가 정의된 DAG의 python 파일들이 있는 폴더
- 스케줄러와 executor 가 읽을 수 있는 dags 경로에 있어야 한다.

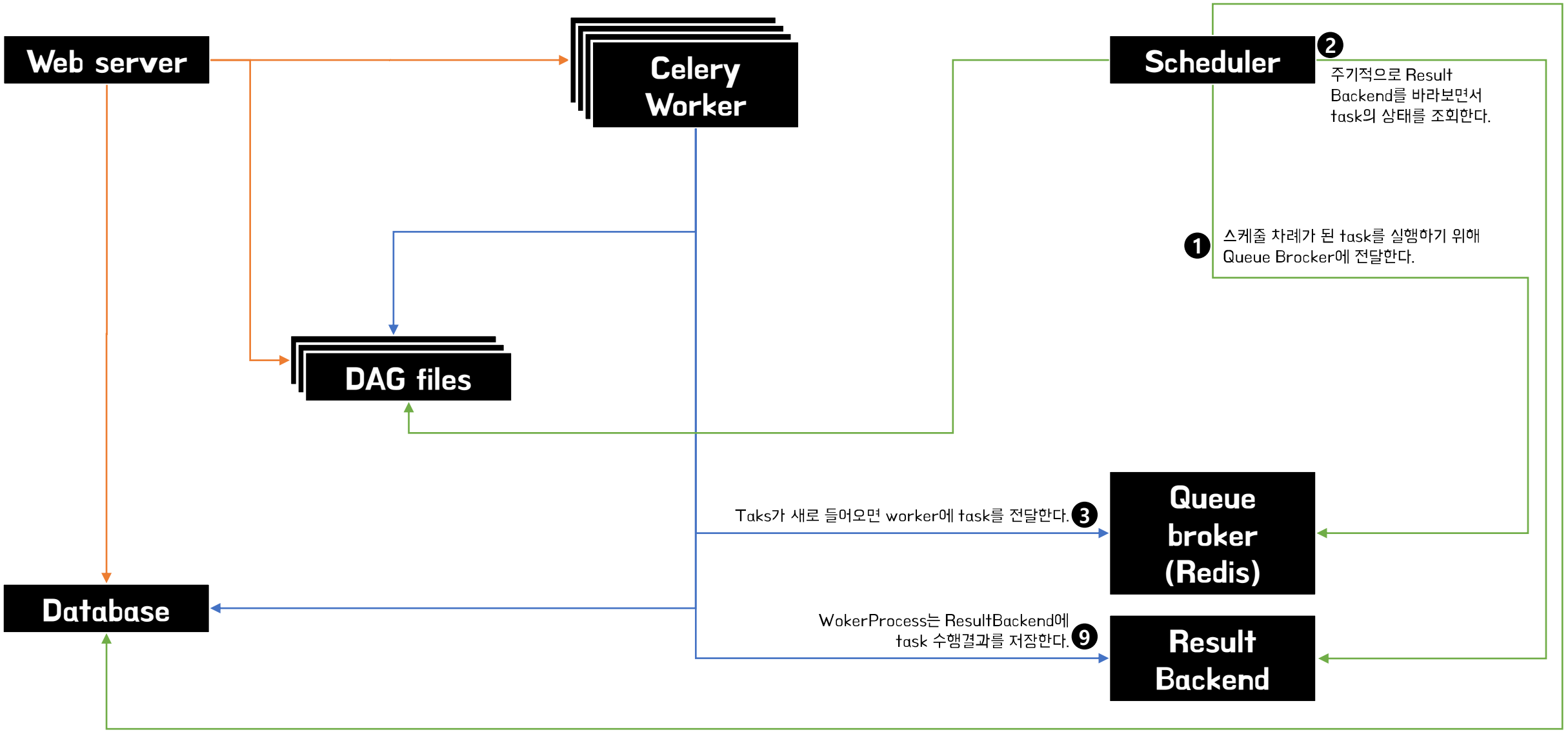
- Metadata Database
  - 작업의 정의, 상태, 실행정보, 결과정보, 로그, audit 등을 관리하는 데이터베이스
  - Scheduler, Executor, Webserver 모두 Metadata Database를 바라본다.

Airflow Component간 Communication (with celery executer)



Airflow Component간 실행 순서 (with celery executer)

4. WorkerProcess는 task하나를 하나의 WorkerChildProcess에 할당한다.
5. WorkerChildProcess는 task handling function을 수행한다.
   그 결과 LocalTaskJobProcess를 생성한다.
6. LocalTaskJobProcess는 TaskRunner를 통해 새로운 프로세스를 실행한다.
7. RawTaskProcess를 수행하고 LocalTaskJobProcess는 그 작업이 끝날때까지 기다린다.
8. WorkerChildProcess는 main process인 workerProcess에 task의 종료를 알리고, 후속 tasks가 가능한다고 정보를 알린다.



Airflow Dag란?

- Airflow에서 실행할 작업들을 순서에 맞게 구성한 WorkFlow를 의미한다.
- Directed Acyclic Graph(비순환 그래프)의 약자이며, DAG를 구성하는 각 작업들을 Task라고 한다.
- DAG는 Task의 관계와 종속성을 반영하여 구조화되어 있다.
- 연결된 화살표의 방향 순서대로 Task를 실행하고, 분기 실행과 병렬 실행이 가능하다.
- 기본적으로 $AIRFLOW_HOME/dags에 저장되며, airflow.cfg파일에서 위치를 수정할 수 있다.
- Dag가 실행되는 호스트는 로컬이 아닐 수 있다.
- 여러 worker를 구성했을 경우 각 worker도 동일한 dags파일을 가지고 있어야 수행될 수 있다.

Airflow Dag 기본 구성

1) 라이브러리 임포트

```
from datetime import datetime, timedelta
from textwrap import dedent

from airflow import DAG
from airflow.operators.bash import BashOperator
```

2) With 절로 DAG 인스턴스 생성

```
with DAG(
    "tutorial",
    default_args={
        "depends_on_past": False,
        "email": ["airflow@example.com"],
        "email_on_failure": False,
        "email_on_retry": False,
        "retries": 1,
        "retry_delay": timedelta(minutes=5),
    },
    description="A simple tutorial DAG",
    schedule=timedelta(days=1),
    start_date=datetime(2021, 1, 1),
    catchup=False,
    tags=["example"],
) as dag:
```

3) Task 구성

```
    t1 = BashOperator(
        task_id="print_date",
        bash_command="date",
    )

    t2 = BashOperator(
        task_id="sleep",
        depends_on_past=False,
        bash_command="sleep 5",
        retries=3,
    )

    t1.doc_md = dedent(
        """ #### Task Documentation"""
    )

    dag.doc_md = __doc__
    dag.doc_md = """This is a documentation placed anywhere"""

    templated_command = dedent(
        """
    {% for i in range(5) %}
        echo "{{ ds }}"
        echo "{{ macros.ds_add(ds, 7)}}"
    {% endfor %}
    """
    )

    t3 = BashOperator(
        task_id="templated",
        depends_on_past=False,
        bash_command=templated_command,
    )
```

4) Task 간 종속성 정의

```
    t1 >> [t2, t3]
```

Airflow Dag 기본 구성

2) With 절로 DAG 인스턴스 생성

```
with DAG(
    dag_id="tutorial",
    default_args={
        "depends_on_past": False,
        "email": ["airflow@example.com"],
        "email_on_failure": False,
        "email_on_retry": False,
        "retries": 1,
        "retry_delay": timedelta(minutes=5),
    },
    description="A simple tutorial DAG",
    schedule=timedelta(days=1),
    start_date=datetime(2021, 1, 1),
    catchup=False,
    tags=["example"],
) as dag:
```

하나의 DAG에는 여러 개의 Task를 정의할 수 있다.
Airflow는 하나의 DAG에 대한 identify를 dag_id와 start_date로 한다. 때문에 한번 scheduler에 등록된 시기부터는 start_date는 변경할 수 없다. 따라서 한번 사용한 dag_id와 같은 start_date는 재사용하지 않는 것이 좋다.

- **dag_id** : DAG의 식별자용 아이디
- **description** : Web ui의 dag 상세에서 dag_id 오른쪽에 뜨는 dag 설명
- **schedule_interval** : DAG 실행 주기
    - Cron expression을 사용해서 정의할 수 있다.
- **start_date** : DAG 실행 시작일자
- **end_date** : DAG 실행을 중지해야 하는 날짜(일반적으로 없음)
- **catchup** : true인 경우 실행 일자가 23.01.01 이더라도 설정한 스케줄에 맞춰서 2021.01.01 부터 실행된다. 스케줄링이 하루단위라면 730개의 스케줄링이 실행하자마자 생긴다.
- **tags** : dag를 분류하고 검색하기 쉽게 할 수 있도록 tag를 붙일 수 있다.

Default_args 목록

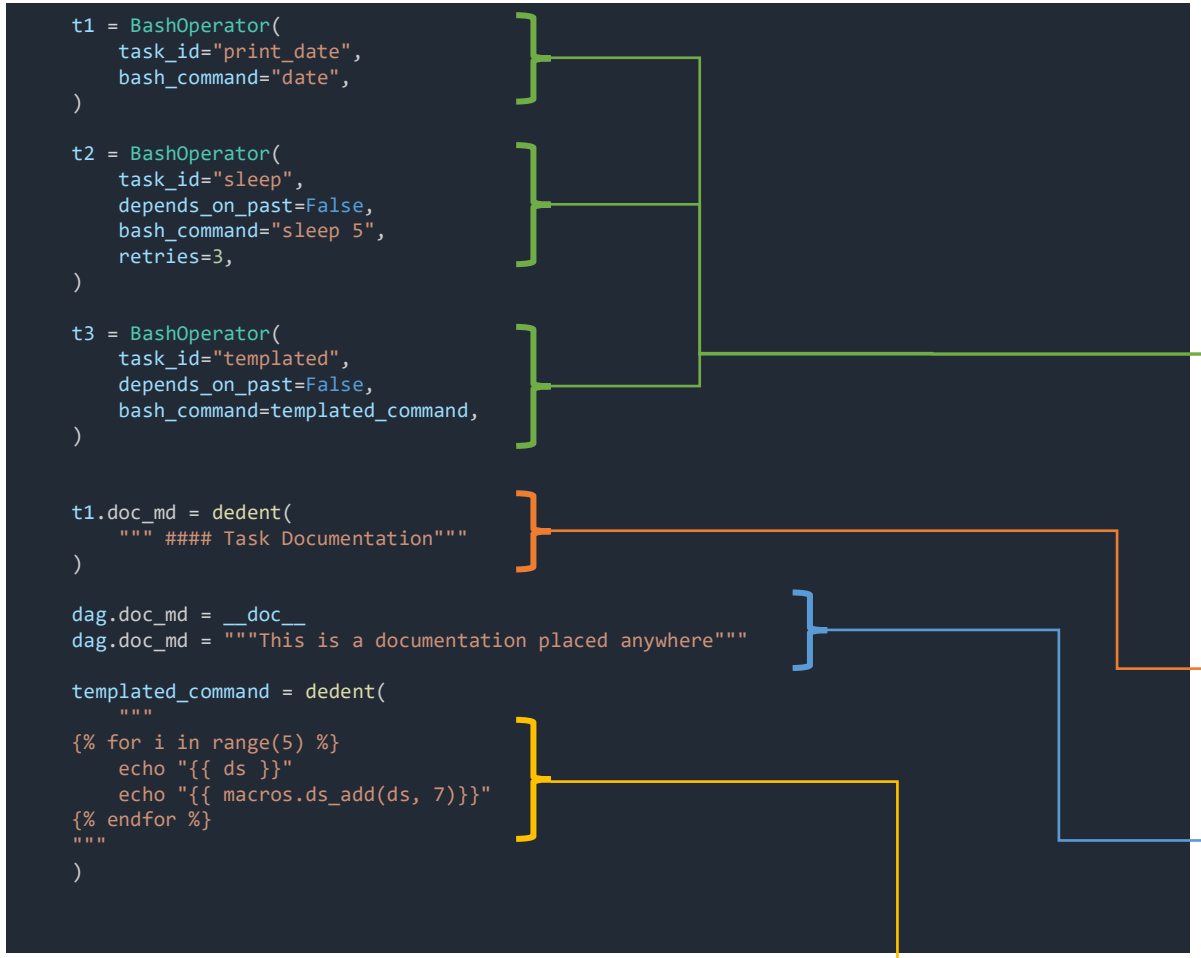
```
    default_args={
        "depends_on_past": False,
        "email": ["airflow@example.com"],
        "email_on_failure": False,
        "email_on_retry": False,
        "retries": 1,
        "retry_delay": timedelta(minutes=5),
        # 'queue': 'bash_queue',
        # 'pool': 'backfill',
        # 'priority_weight': 10,
        # 'end_date': datetime(2016, 1, 1),
        # 'wait_for_downstream': False,
        # 'sla': timedelta(hours=2),
        # 'execution_timeout': timedelta(seconds=300),
        # 'on_failure_callback': some_function, # or list of functions
        # 'on_success_callback': some_other_function, # or list of functions
        # 'on_retry_callback': another_function, # or list of functions
        # 'sla_miss_callback': yet_another_function, # or list of functions
        # 'trigger_rule': 'all_success'
    },
```

주요 설정
default_args는 DAG의 파라미터로 딕셔너리에서 풀어서 작성해도 되지만 이와 같이 여러 곳에서 같은 args를 사용하기 위해 딕셔너리화 하여 전달 할 수도 있음

- **depends_on_past** : 이전 스케줄링된 (DAG스케줄 기준) task가 완료 되어야 하는지 여부
- **retries** : 실패 시 자동 재처리 횟수. 최소 1, 보통 3회 권장
- **retry_delay** : 실패 시 다음 재시도까지 기다리는 시간
- **queue** : 해당 task를 넣을 queue, 설정하지 않으면 airflow.cfg의 default queue
    - Worker 노드를 특정하고 싶을 때 사용
- **wait_for_downstream** : 이전에 스케줄링 된 task의 downstream 작업들이 완료되어야 하는지 여부
- **execution_timeout** : 실행시간의 timeout
    - 비정상적으로 오래걸려서 hamg 걸리는 작업을 방지
    - Celery 버전과 설정에 영향을 받을 수 있다.
- **XXX_callback** : 해당 조건하에서 수행할 수 있는 callback 함수. 특별한 notify 또는 자원이나 변수 정리 등에 사용한다.
- **task_concurrency** : 같은 task에 대해 동시에 active run 상태를 허용하는 task 수준에서 concurrency

Airflow Dag 기본 구성

3) Task 구성



```
t1 = BashOperator(
    task_id="print_date",
    bash_command="date",
)

t2 = BashOperator(
    task_id="sleep",
    depends_on_past=False,
    bash_command="sleep 5",
    retries=3,
)

t3 = BashOperator(
    task_id="templated",
    depends_on_past=False,
    bash_command=templated_command,
)


t1.doc_md = dedent(
    """ #### Task Documentation"""
)

dag.doc_md = __doc__
dag.doc_md = """This is a documentation placed anywhere"""

templated_command = dedent(
    """
{% for i in range(5) %}
    echo "{{ ds }}"
    echo "{{ macros.ds_add(ds, 7)}}"
{% endfor %}
"""
)
```

DAG의 with절 안에 task를 구성한다.
왼쪽의 예시는 t1, t2, t3로 task 인스턴스를 담을 변수명을 임의로 정한 것이며 이는 작성자가 변경 가능하다.
각 task 인스턴스는 task의 단위이며 작업할 내용을 담고 있는 Operator를 가지고 있다.

Operator

Operator는 airflow가 완료될 task 단위를 정의한다.
모든 Operator는 Base Operator를 상속하고 있으며, 여기에는 airflo에서 task를 실행하는데 필요한 모든 인수가 포함되어 있다.
가장 인기있는 Operator로는 PythonOperator, BashOperator, KubernetesPodOperator 등이 있다.

- **PythonOperator** : 파이썬 함수를 실행시키기 위한 Operator
- **BashOperator** : Bash 명령어를 실행시키기 위한 Operator

- **task_id** : task의 id. Web ui의 시각화에서 사용되며 airflow에서 task를 식별할 때에도 사용된다.
- 이외 retries 등 특정 인수의 우선순위 규칙
    - 1. task를 작성하면서 명시적으로 전달한 argument
    - 2. dag를 작성하면서 전달한 default_args에 존재하는 값
    - 3. operato의 default 값(존재하는 경우)

Task에 코멘트 달기 (필수 x)

Task_id.doc_md = dedent() 를 선언하고 괄호 안에 문자열로 Markdown형식으로 코멘트를 작성하면 task 상세 페이지에서 해당 내용을 확인할 수 있다.

DAG 문서에 코멘트 달기 (필수 x)

Dag.doc_md = __doc__ 를 명시하고 dag.doc_md = “”에 문자열로 Markdown형식으로 코멘트를 작성하면 dag 상세 화면 어디에서든 해당 코멘트를 확인할 수 있다.

Templating with Jinja

Airflow는 파이프라인 작성자에게 일련의 기본 제공 매개 변수 및 매크로를 제공한다. 또한 airflow는 파이프라인 작성자가 자체 매개변수 매크로 및 템플릿을 정의할 수 있는 후크를 제공한다.
왼쪽 예시는 코드 로직을 {% %}사이에 두었고, {{ ds }}같은 파라미터를 참고하고, {{macros.ds_add(ds, 7) }} 같은 함수를 참조한다.
{{ ds }}는 today’s date stamp를 반환한다.

Airflow Dag 기본 구성

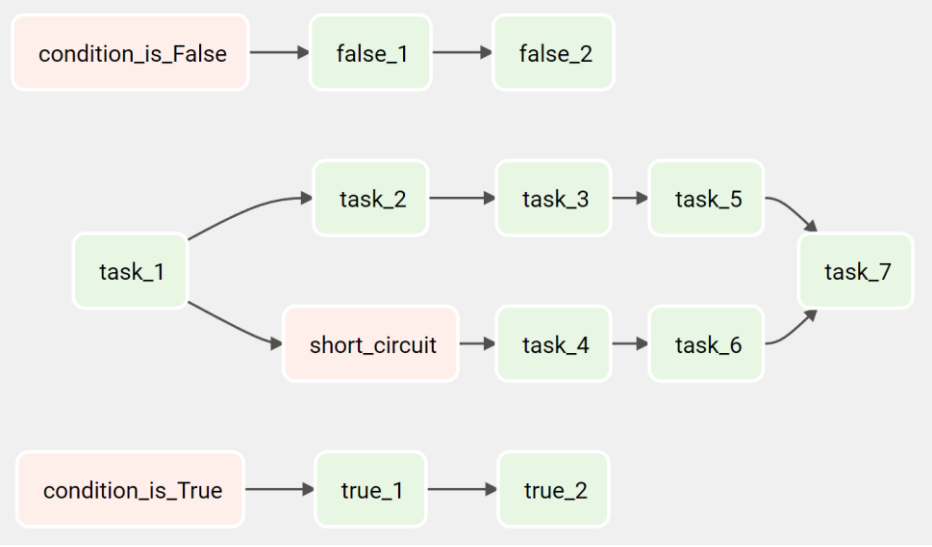
4) Task 간 종속성 정의

```
    t1 >> [t2, t3]
```

Task 간에 종속성(방향성)을 간단하게 >> 로 정의할 수 있으며 아래와 같이 복잡한 방식으로 작성하는 것도 가능하다.

example_short_circuit_operator의 종속성 예시

```
    cond_true = ShortCircuitOperator(
        task_id="condition_is_True",
        python_callable=lambda: True,
    )

    cond_false = ShortCircuitOperator(
        task_id="condition_is_False",
        python_callable=lambda: False,
    )

    ds_true = [EmptyOperator(task_id="true_" + str(i)) for i in [1, 2]]
    ds_false = [EmptyOperator(task_id="false_" + str(i)) for i in [1, 2]]

    chain(cond_true, *ds_true)
    chain(cond_false, *ds_false)

    [task_1, task_2, task_3, task_4, task_5, task_6] = [
        EmptyOperator(task_id=f"task_{i}") for i in range(1, 7)
    ]

    task_7 = EmptyOperator(task_id="task_7", trigger_rule=TriggerRule.ALL_DONE)

    short_circuit = ShortCircuitOperator(
        task_id="short_circuit", ignore_downstream_trigger_rules=False, python_callable=lambda:
False
    )

    chain(task_1, [task_2, short_circuit], [task_3, task_4], [task_5, task_6], task_7)
```



Airflow 기타 기본 개념 스케줄링

Ariflow DAG를 작성할 때 schedule_interval 인수를 사용하여 스케줄 간격을 정의할 수 있다.
Default 값은 None이며 이 경우 스케줄러는 작업을 예약 실행하지 않고 UI 또는 API를 통해서 수동으로 trigger시켜 실행시킬 수 있다.

- 1. 스케줄 간격을 의미하는 매크로 사용

@once : 1회만 실행되도록 스케줄
@hourly : 매시간 변경 시 1회 실행
@daily : 매일 자정에 1회 실행
@weekly : 매주 일요일 자정에 1회 실행
@monthly : 매월 1일 자정에 1회 실행
@yearly : 매년 1월 1일 자정에 1회 실행

- 2. 빈도 기반 스케줄링 (datetime.timedelta)

```
import datetime as dt

dag=DAG(
    dag_id="dag_id",
    schedule_interval=dt.timedelta(days=3),
    start_date=dt.datetime(year=2023, month=1, day=1),
    end_date=dt.datetime(year=2023, month=1, day=5),
)
```

이렇게 설정하면 시작 시간으로부터 3일마다(2023.01.04, 07, 10 ...) 실행된다

- 3. Cron 기반 스케줄링 사용

```
┌──────── 분 (0 ~ 59)
│ ┌────── 시간 (0 ~ 23)
│ │ ┌──── 일 (1 ~ 31)
│ │ │ ┌── 월 (1 ~ 12)
│ │ │ │ ┌ 요일 (0 ~ 6) (일요일 ~ 토요일; 일부 시스템에서 7은 일요일이다.)
* * * * *
```

- 0 * * * * = 매시간(정시에 실행)
- 0 0 * * * = 매일 (자정에 실행)
- 0 0 * * 0 = 매주 (일요일 자정에 실행)
- 0 0 1 * * = 매월 1일 자정에 실행
- 45 23 * * SAT = 매주 토요일 23시 45분에 실행
- 0 0 * * MON, WED, FRI = 매주 월, 화, 금요일 자정에 실행
- 0 0,12 * * * = 매일 자정 및 오후 12시에 실행

구성 요소

field	required	Allowed values	Allowed special characters
Minutes	Yes	0-59	* , -
Hours	Yes	0-23	* , -
Day of month	Yes	01월 31일	* , -
Month	Yes	1-12 or JAN-DEC	* , -
Day of week	Yes	0-6 or SUN-SAT	* , -

Airflow 기타 기본 개념 원자성, 멱등성

- 원자성

Airflow에서 원자성 트랜잭션은 모두 발생하거나 전혀 발생하지 않는, 나눌 수 없고 돌이킬 수 없는 일련의 데이터베이스와 같은 작업으로 간주한다. 따라서 Airflow의 task는 성공적으로 수행하여 적절한 결과를 생성하거나 시스템 상태에 영향을 미치지 않고 실패하도록 정의한다.

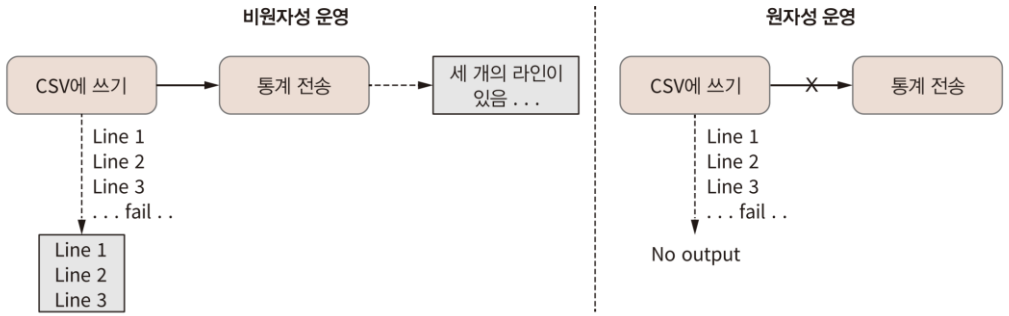


그림 3.9 원자성은 모든 것이 완료되거나 완료되지 않도록 보장해야 합니다. 절반의 태스크가 진행되지 않아, 결과적으로 잘못된 결과가 발생하지 않습니다.

- 멱등성

동일한 입력으로 동일한 task를 여러 번 호출해도 결과에 영향이 없어야 한다. 즉 입력 변경 없이 task를 다시 실행해도 전체 결과가 변경되지 않아야 한다는 것이다.

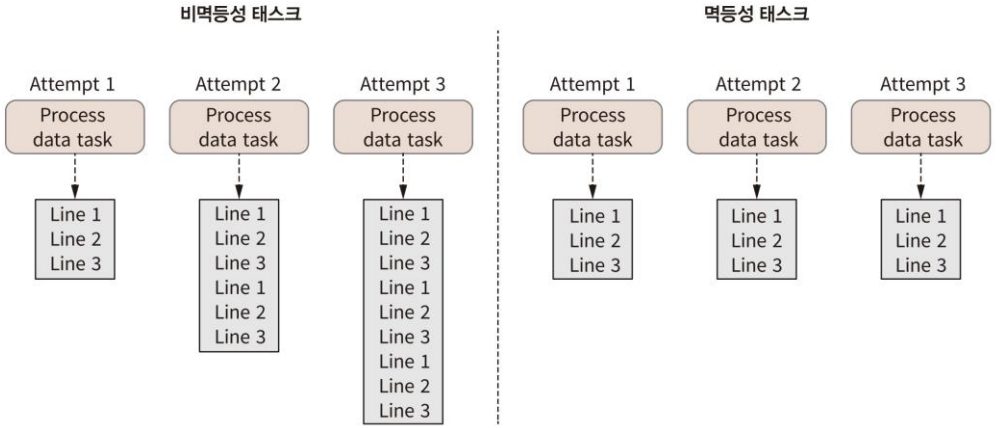


그림 3.10 멱등성이 보장되는 태스크는 실행 횟수에 관계없이 동일한 결과를 생성합니다. 멱등성은 일관성과 장애 처리를 보장합니다.

Airflow 기타 기본 개념 Jinja template, Context

Dag에서 모든 Operator 인수가 jinja template에서 사용될 수 있는 것은 아님
Jinja에서 템플릿 화 가능한 속성 리스트에 포함되어 있어야 한다.
모든 Operator의 template)fields 속성에 의해 설정된다.
아래 내용은 PythonOperator로부터 python_callable함수에 전달된 context를 출력한 내용이다.

```
{
    'conf': <***.configuration.AirflowConfigParser object at 0x7f17e15ccb10>,
    'dag': <DAG: listing_4_01_03>,
    'dag_run': <DagRun listing_4_01_03 @ 2023-06-20 11:26:34.068525+00:00: manual__2023-06-20T11:26:34.068525+00:00, state:running, queued_at: 2023-06-20 11:26:34.085467+00:00. externally triggered: True>,
    'data_interval_end': DateTime(2023, 6, 20, 11, 0, 0, tzinfo=Timezone('UTC')),
    'data_interval_start': DateTime(2023, 6, 20, 10, 0, 0, tzinfo=Timezone('UTC')),
    'ds': '2023-06-20',
    'ds_nodash': '20230620',
    'execution_date': <Proxy at 0x7f17b7b580f0 with factory functools.partial(<function lazy_mapping_from_context.<locals>._deprecated_proxy_factory at 0x7f17b7ba0ef0>, 'execution_date', DateTime(2023, 6, 20, 11, 26, 34, 68525,
tzinfo=Timezone('UTC')))>,
    'expanded_ti_count': None,
    'inlets': [],
    'logical_date': DateTime(2023, 6, 20, 11, 26, 34, 68525, tzinfo=Timezone('UTC')),
    'macros': <module '***.macros' from '/home/***/.local/lib/python3.7/site-packages/***/macros/__init__.py'>,
    'next_ds': <Proxy at 0x7f17b7b58140 with factory functools.partial(<function lazy_mapping_from_context.<locals>._deprecated_proxy_factory at 0x7f17b7ba0ef0>, 'next_ds', '2023-06-20')>,
    'next_ds_nodash': <Proxy at 0x7f17b7b58190 with factory functools.partial(<function lazy_mapping_from_context.<locals>._deprecated_proxy_factory at 0x7f17b7ba0ef0>, 'next_ds_nodash', '20230620')>,
    'next_execution_date': <Proxy at 0x7f17b7b58230 with factory functools.partial(<function lazy_mapping_from_context.<locals>._deprecated_proxy_factory at 0x7f17b7ba0ef0>, 'next_execution_date', DateTime(2023, 6, 20, 11, 26, 34, 68525,
tzinfo=Timezone('UTC')))>,
    'outlets': [],
    'params': {},
    'prev_data_interval_start_success': DateTime(2023, 6, 20, 10, 0, 0, tzinfo=Timezone('UTC')),
    'prev_data_interval_end_success': DateTime(2023, 6, 20, 11, 0, 0, tzinfo=Timezone('UTC')),
    'prev_ds': <Proxy at 0x7f17b7b58280 with factory functools.partial(<function lazy_mapping_from_context.<locals>._deprecated_proxy_factory at 0x7f17b7ba0ef0>, 'prev_ds', '2023-06-20')>,
    'prev_ds_nodash': <Proxy at 0x7f17b7b582d0 with factory functools.partial(<function lazy_mapping_from_context.<locals>._deprecated_proxy_factory at 0x7f17b7ba0ef0>, 'prev_ds_nodash', '20230620')>,
    'prev_execution_date': <Proxy at 0x7f17b7b58320 with factory functools.partial(<function lazy_mapping_from_context.<locals>._deprecated_proxy_factory at 0x7f17b7ba0ef0>, 'prev_execution_date', DateTime(2023, 6, 20, 11, 26, 34, 68525,
tzinfo=Timezone('UTC')))>,
    'prev_execution_date_success': <Proxy at 0x7f17b7b58370 with factory functools.partial(<function lazy_mapping_from_context.<locals>._deprecated_proxy_factory at 0x7f17b7ba0ef0>, 'prev_execution_date_success', DateTime(2023, 6, 20, 11, 25, 35,
698997, tzinfo=Timezone('UTC')))>,
    'prev_start_date_success': DateTime(2023, 6, 20, 11, 25, 36, 98642, tzinfo=Timezone('UTC')),
    'run_id': 'manual__2023-06-20T11:26:34.068525+00:00',
    'task': <Task(PythonOperator): print_context_listing_4_03>,
    'task_instance': <TaskInstance: listing_4_01_03.print_context_listing_4_03 manual__2023-06-20T11:26:34.068525+00:00 [running]>,
    'task_instance_key_str': 'listing_4_01_03__print_context_listing_4_03__20230620',
    'test_mode': False,
    'ti': <TaskInstance: listing_4_01_03.print_context_listing_4_03 manual__2023-06-20T11:26:34.068525+00:00 [running]>,
    'tomorrow_ds': <Proxy at 0x7f17b7b583c0 with factory functools.partial(<function lazy_mapping_from_context.<locals>._deprecated_proxy_factory at 0x7f17b7ba0ef0>, 'tomorrow_ds', '2023-06-21')>,
    'tomorrow_ds_nodash': <Proxy at 0x7f17b7b58410 with factory functools.partial(<function lazy_mapping_from_context.<locals>._deprecated_proxy_factory at 0x7f17b7ba0ef0>, 'tomorrow_ds_nodash', '20230621')>,
    'triggering_dataset_events': <Proxy at 0x7f17b7befeb0 with factory <function TaskInstance.get_template_context.<locals>.get_triggering_events at 0x7f17b7bdc290>>,
    'ts': '2023-06-20T11:26:34.068525+00:00',
    'ts_nodash': '20230620T112634',
    'ts_nodash_with_tz': '20230620T112634.068525+0000',
    'var': {'json': None, 'value': None},
    'conn': None,
    'yesterday_ds': <Proxy at 0x7f17b7b58460 with factory functools.partial(<function lazy_mapping_from_context.<locals>._deprecated_proxy_factory at 0x7f17b7ba0ef0>, 'yesterday_ds', '2023-06-19')>,
    'yesterday_ds_nodash': <Proxy at 0x7f17b7b584b0 with factory functools.partial(<function lazy_mapping_from_context.<locals>._deprecated_proxy_factory at 0x7f17b7ba0ef0>, 'yesterday_ds_nodash', '20230619')>,
    'templates_dict': None
}
```

Airflow 기타 기본 개념 Jinja template, Context

주요 task context 정리

key	description	example	example
conf	Airflow 구성에 대해 접근할 수 있다.	airflow.configuration. AirflowConfigParser object	<***.configuration.AirflowConfigParser object at 0x7f17e15ccb10>
dag	현재 DAG 객체	DAG object	<DAG: listing_4_01_03>
dag_run	현재 DagRun 객체	DagRun object	<DagRun listing_4_01_03 @ 2023-06-20 11:26:34.068525+00:00: manual__2023-06-20T11:26:34.068525+00:00, state:running, queued_at: 2023-06-20 11:26:34.085467+00:00. externally triggered: True>
ds	%Y-%m-%d 형식의 excution_date	'2023-06-20'	'2023-06-20'
ds_nodash	%Y%m%d 형식의 excution_date	'20230620'	'20230620'
execution_date	태스크 스케줄 간격의 시작 날짜/시간	pendulum.datetime.DateTime object	<Proxy at 0x7f17b7b580f0 with factory functools.partial(<function lazy_mapping_from_context.<locals>._deprecated_proxy_factory at 0x7f17b7ba0ef0>, 'execution_date', DateTime(2023, 6, 20, 11, 26, 34, 68525, tzinfo=Timezone('UTC')))>
inlets	task.inlets의 약어 데이터 계보에 대한 입력 데이터 소스를 추적하는 기능	[]	[]
macros	airflow.macros 모듈	macro module	<module '***.macros' from '/home/***/.local/lib/python3.7/site-packages/***/macros/__init__.py'>
next_ds	%Y-%m-%d 형식의 다음 스케줄 간격 (=현재 스케줄 간격의 끝)의 execution_date	'2023-06-20'	<Proxy at 0x7f17b7b58140 with factory functools.partial(<function lazy_mapping_from_context.<locals>._deprecated_proxy_factory at 0x7f17b7ba0ef0>, 'next_ds', '2023-06-20')>
next_ds_nodash	%Y%m%d 형식의 다음 스케줄 간격 (=현재 스케줄 간격의 끝)의 execution_date	'20230620'	<Proxy at 0x7f17b7b58190 with factory functools.partial(<function lazy_mapping_from_context.<locals>._deprecated_proxy_factory at 0x7f17b7ba0ef0>, 'next_ds_nodash', '20230620')>
next_execution_date	태스크의 다음 스케줄 간격의 시작 datetime(=현재 스케줄 간격의 끝)	pendulum.datetime.DateTime object	<Proxy at 0x7f17b7b58230 with factory functools.partial(<function lazy_mapping_from_context.<locals>._deprecated_proxy_factory at 0x7f17b7ba0ef0>, 'next_execution_date', DateTime(2023, 6, 20, 11, 26, 34, 68525, tzinfo=Timezone('UTC')))>
outlets	task.outlets의 약어 데이터 계보 lineage에 대한 출력 데이터 소스를 추적하는 기능	[]	[]
params	태스크 콘텍스트에 대한 사용자 제공 변수	{}	{}
prev_ds	%Y-%m-%d 형식의 이전 스케줄 간격의 execution_date	'2023-06-20'	<Proxy at 0x7f17b7b58280 with factory functools.partial(<function lazy_mapping_from_context.<locals>._deprecated_proxy_factory at 0x7f17b7ba0ef0>, 'prev_ds', '2023-06-20')>
prev_ds_nodash	%Y%m%d 형식의 이전 스케줄 간격의 execution_date	'20230620'	<Proxy at 0x7f17b7b582d0 with factory functools.partial(<function lazy_mapping_from_context.<locals>._deprecated_proxy_factory at 0x7f17b7ba0ef0>, 'prev_ds_nodash', '20230620')>
prev_execution_date	태스크 이전 스케줄 간격의 시작 datetime	pendulum.datetime.DateTime object	<Proxy at 0x7f17b7b58320 with factory functools.partial(<function lazy_mapping_from_context.<locals>._deprecated_proxy_factory at 0x7f17b7ba0ef0>, 'prev_execution_date', DateTime(2023, 6, 20, 11, 26, 34, 68525, tzinfo=Timezone('UTC')))>
prev_execution_date_success	동일한 태스크의 마지막으로 성공적으로 완료된 실행의 시작 datetime(과거에만 해당)	pendulum.datetime.DateTime object	<Proxy at 0x7f17b7b58370 with factory functools.partial(<function lazy_mapping_from_context.<locals>._deprecated_proxy_factory at 0x7f17b7ba0ef0>, 'prev_execution_date_success', DateTime(2023, 6, 20, 11, 25, 35, 698997, tzinfo=Timezone('UTC')))>
prev_start_date_susccess	동일한 태스크의 마지막으로 성공적으로 시작된 날짜와 시간(과거에만 해당)	pendulum.datetime.DateTime object	DateTime(2023, 6, 20, 11, 25, 36, 98642, tzinfo=Timezone('UTC'))
run_id	DagRun의 run_id (일반적으로 접두사 + datetime으로 구성된 키)	'manual__2019-0101T00:00:00+00:00'	'manual__2023-06-20T11:26:34.068525+00:00'
task	현재 오퍼레이터	PythonOperator object	<Task(PythonOperator): print_context_listing_4_03>
task_instance	현재 TaskInstance 객체	TaskInstance object	<TaskInstance: listing_4_01_03.print_context_listing_4_03 manual__2023-06-20T11:26:34.068525+00:00 [running]>
task_instance_key_str	현재 TaskInstance의 고유 식별자 ({dag_id}__{task_id}__{ds_nodash})	'dag_id__task_id__20230620'	'listing_4_01_03__print_context_listing_4_03__20230620'
templates_dict	태스크 곤텍스트에 대한 사용자 제공 변수	{}	위에서 프린트한 내용에는 없음
test_mode	Airflow가 테스크 모드에서 실행중인지 여부(구성 속성)	FALSE	FALSE
ti	task_instance와 동일한 현재 TaskInstance 객체	TaskInstance object	<TaskInstance: listing_4_01_03.print_context_listing_4_03 manual__2023-06-20T11:26:34.068525+00:00 [running]>
tomorrow_ds	ds(실행 시간)에서 1일을 더함	'2023-06-21'	<Proxy at 0x7f17b7b583c0 with factory functools.partial(<function lazy_mapping_from_context.<locals>._deprecated_proxy_factory at 0x7f17b7ba0ef0>, 'tomorrow_ds', '2023-06-21')>
tomorrow_ds_nodash	ds_nodash에서 1일을 더함	'20230621'	<Proxy at 0x7f17b7b58410 with factory functools.partial(<function lazy_mapping_from_context.<locals>._deprecated_proxy_factory at 0x7f17b7ba0ef0>, 'tomorrow_ds_nodash', '20230621')>
ts	ISO8601 포멧에 따른 execution_date	'2023-06-20T00:00:00+00:00'	'2023-06-20T11:26:34.068525+00:00'
ts_nodash	%Y%m%d%T%H%M%S 형식의 execution_date	'20230620T000000'	'20230620T112634'
ts_nodash_with_tz	시간 정보가 있는 ts_nodash	'20230620T000000+0000'	'20230620T112634.068525+0000'
var	Airflow 변수를 처리하기 위한 헬퍼 개체 Helpers object	{}	{'json': None, 'value': None}
yesterday_ds	ds(실행시간) 1일을 뺌	'2023-06-19'	<Proxy at 0x7f17b7b58460 with factory functools.partial(<function lazy_mapping_from_context.<locals>._deprecated_proxy_factory at 0x7f17b7ba0ef0>, 'yesterday_ds', '2023-06-19')>
yesterday_ds_nodash	ds_nodash에서 1일을 뺌	'20230619'	<Proxy at 0x7f17b7b584b0 with factory functools.partial(<function lazy_mapping_from_context.<locals>._deprecated_proxy_factory at 0x7f17b7ba0ef0>, 'yesterday_ds_nodash', '20230619')>

Airflow 기타 기본 개념 Jinja template, Context

Jinja template 인수 사용방법

```
# callable 함수 내부에서 템플릿 문자열을 처리하기
def _get_data(execution_date, **context):
    year, month, day, hour, *_ = execution_date.timetuple()
    sample_url = f"{year}/{year}-{month:0>2}/pageviews-{year}{month:0>2}{day:0>2}-{hour:0>2}0000.gz"
    print(sample_url)

get_data = PythonOperator(
              task_id="get_data",
              python_callable=_get_data,
              dag=dag
           )


# 템플릿 문자열을 호출하여 callable 함수에 명시적으로 넣기
def _get_data(year, month, day, hour, output_path, **_):
    sample_url = f"{year}/{year}-{month:0>2}/pageviews-{year}{month:0>2}{day:0>2}-{hour:0>2}0000.gz"
    print(sample_url)


get_data = PythonOperator(
    task_id="get_data",
    python_callable=_get_data,
    op_kwargs={
        "year": "{{ execution_date.year }}",
        "month": "{{ execution_date.month }}",
        "day": "{{ execution_date.day }}",
        "hour": "{{ execution_date.hour }}",
        "output_path": "/tmp/wikipageviews.gz",
    },
    dag=dag,
)
```

Airflow 기타 기본 개념　Parallelism(병렬성), Concurrency(동시성) 설정

- Airflow.cfg 에서의 설정

Core.parallelism : 워커가 몇 개이든 상관 없이, 한번에 실행할 수 있는 task의 개수가 몇 개이든 상관 없이 한번에 실행 시킬 수 있는 총 task의 개수
Core.dag_concurrency : 하나의 dag당 active 하게 동시에 실행 시킬 수 있는 task의 개수. Dag에서 이를 특별히 설정하지 않는한 airflow.cfg의 설정에 영향을 받는다.
Core.non_pooled_task_slot_count : number of task slots allowcated to tasks not running in a pool
Core.max_active_runs_per_dag : 한번에 active(running 상태인 dag)일 수 있는 dag의 개수
Scheduler.max_threads : 스케줄러가 가질 수 있는 스레드 개수. 스케줄러가 설치되어 있는 서버 사양에 영향을 받으며, 한번에 active상태가 될 수 있는 dag의 개수에 영향을 미친다.
Celery.worker_concurrency : 하나의 워커가 가져갈 수 있는 task의 개수. 기본적으로 워커가 설치되어 있는 서버 사양에 영향을 받는다. 총 개수를 코어 수 이하로 맞춰주는걸 권장함
Celery.sync_parallelism : task의 상태 싱크를 맞춰주는 celery excutor의 프로세스 수

- DAG에서의 설정

concurrency : 모든 active run 상태인 dags 들 하위에서 실행 가능한 최대 task instance의 수. 따로 설정하지 않으면 airflow.cfg의 core.dag_concurrency를 사용한다.
: dag의 개수가 4개이고 각 dag당 스케줄링 되어 실행시켜야만 하는 task의 개수가 4일 때 concurrency가 7dlfkaus 총 16개의 task 중에 7개만 스케줄링 되고 나머지 9개는 대기한다.
max_active_runs : 해당 dag의 최대 active run 상태가 가능한 DAG(한번에 스케줄링 된 DAG)의 수.
: 해당 값이 max에 도달하면 스케줄러는 다음 dag를 스케줄링 하지 않는다.
: active(running 상태인 dag)상태가 될 수 있는 dag의 총 개수의 합
: 3개의 task가 있는 dag가 여러 개 있을 때 max_active_runs를 4개로 둔다면 concurrency는 12로 두는게 합당하다.

- Operator에서의 설정

pool args를 사용하여 해당 task가 pool의 개수 만큼만 실행되도록 설정할 수 있다. Pool 목록은 pool이름을 지정하고 여러 allocated slots을 할당하여 UI(Menu -> admin -> pool)에서 관리된다. 그런 다음 task를 생성할 때 pool매개변수를 사용하여 작업을 기존 pool중 하나와 연결할 수 있다. Slot이 채워지는 동안 task는 평소와 같이 예약된다. Task에서 사용하는 slot의 수는 pool_slots로 구성할 수 있다. 용량에 도달하게 되면 실행 가능한 task는 대기열에 들어가고 해당 상태가 UI에 표시 된다. Slot이 확보되면 대기중인 ask는 task 및 하위 task의 우선순위 가중치를 기반으로 실행되기 시작한다. Task 풀이 지정되지 않은 경우 task의 기본 풀 default_pool에 할당된다. Default_pool은 128개의 slot으로 초기화 되어 있으며 UI 또는 CLI를 통해 수정할 수는 있지만 삭제할 수는 없다.
실무에서 사용하는 대표적인 사례는 Hadoop(또는 Yarn, Spark 등) 클러스터에 접근할 수 있는 하나의 Operator를 제공하고, 다른 해당 Operator에 대해서 pool을 지정한다. 여러 작업자가 무분별하게 작업한 수 많은 DAG들이 수행될 때 Hadoop 클러스터를 잘못 사용하거나 자원을 지나치게 많이 사용하는 것을 방지할 수 있다.

Airflow 기타 기본 개념 Variable, Connection, XCOM

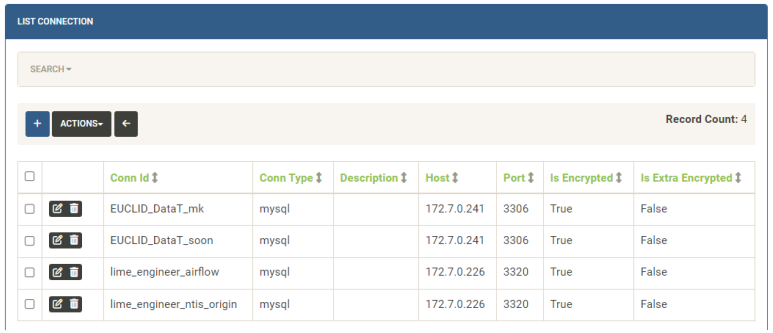
```python
from airflow.models import Variable

# Normal call style
foo = Variable.get("foo")

# Auto-deserializes a Json value
bar = Variable.get("bar", deserialize_json=True) # JSON으로 가져옴

# Returns the value of default_var (None) if the variable is not set
baz = Variable.get("baz", default_var=None)

# Raw value
{{ var.value.<variable_name> }}

# Auto-deserializes Json value
{{ var.json.<variable_name> }}
```

- Variable

Menu -> admin -> variable에서 추가, 삭제할 수 있다.
Task에서 참조할 수 있는 클러스터의 변수이다.
보통 환경에 따라 url이 달라지거나 옵션이 달라지는 경우, DAG 전체에서 공통적으로 관리해야 하는 변수, 환경변수, url에 대한 기본 PATH를 설정하는 데 쓰인다.

- Connection

Database 등의 외부 요소의 연결 + 로그인 정보를 저장하고 task에서 가져와서 사용할 수 있다.
Menu -> admin -> Connection에서 관리 가능하다.
Conn_id로 해당 접속정보를 가져와서 사용할 수 있다.




```python
def _xcom_push(**context):
    model_id = str(uuid.uuid4())
    context["task_instance"].xcom_push(key="model_id", value=model_id)

push_task = PythonOperator(
    task_id="push_task",
    python_callable=_xcom_push
)

def _xcom_pull(**context):
    model_id=context["task_instance"].xcom_pull(
        task_ids="push_task", key="model_id"
    )
    print(f"pull task uuid : {model_id}")

pull_task = PythonOperator(
    task_id="pull_task",
    python_callable=_xcom_pull
)
```

- XCOM

XCOM은 물리적으로 서로 다른 환경에서 동작하는 task들 사이에서 데이터를 주고 받을 때 사용한다.
Xcom은 키와 task_id 및 dag_id로 식별된다.
Xcom으로 큰 값을 전달하는데 절대 사용해서는 안된다. 중간에 데이터 유실의 가능성이 존재한다.
Menu -> Admin -> Xcom 에서 확인 가능

Xcom은 메타 스토어에 저장되며 크기 제한이 있다.
SQLite BLOB유형 2gb 제한
PostgreSQL BYTEA유형 1gb 제한
MySQL BLOB유형 64kb 제한

Airflow 현재 배포 구성 https://gitlab.euso.kr/data_visualization/lime_flow

- Airflow에서 공식으로 제공해주는 docker-compose.yml 파일을 custom하여 재구성

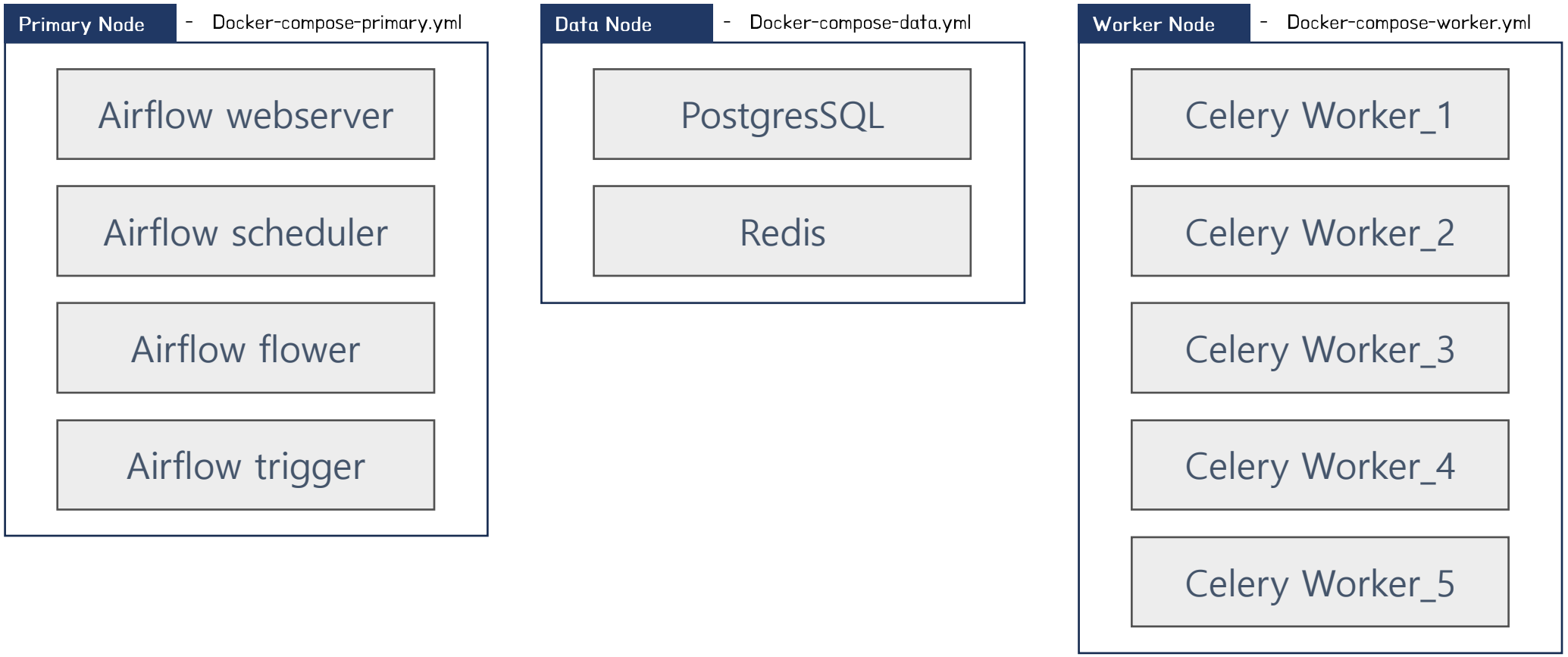
- Dockerfile for custom airflow

```
FROM apache/airflow:slim-2.6.1-python3.8

USER root
RUN sudo apt-get update \
  && apt-get install -y --no-install-recommends \
         vim gcc git psmisc direnv\
         default-libmysqlclient-dev libpq-dev \
  && apt-get autoremove -yqq --purge \
  && apt-get clean \
  && rm -rf /var/lib/apt/lists/*
RUN mkdir -p /opt/settings
USER airflow

COPY ./settings /opt/settings

RUN pip install "apache-airflow[celery, postgres, mysql, redis,
crypto, statsd]==2.6.1" --constraint
"https://raw.githubusercontent.com/apache/airflow/constraints-
2.6.1/constraints-3.8.txt"
RUN pip install -r /opt/settings/requirements.txt
```

- Custom 내용
  - Custom airflow image
    - docker-compose.yml에 있는 airflow image는 우리가 필요한 서비스 외에 60여가지 이상의 provider들이 함께 설치되어 있다. 따라서 가장 가벼운 slim 버전을 다운받아 필요한 서비스만 설치하는 방식으로 dockerfile 을 구성하였다.

  - Primary, worker, data 분리
    - Airflow에 필요한 서비스를 분리한 이유는 각각 확장과 안정성 때문이다.
    - Primary node는 스케줄러와 트리거, 웹서버가 구동되고 있기 때문에 worker로 인해 해당 서비스가 다운되는 것을 방지하기 위해 분리했다.
    - Worker node는 celery로 구성되어 있기 때문에 worker가 많이 필요한 시점에 자유로운 스케일 업을 위해 따로 분리했다.
    - Data node에는 airflow 에서 사용하는 meta data 와 task 의 queue 역할을 하는 redis가 설치되어 있어 primary node 와 마찬가지로 airflow에서 구동하는 task 에 의해 서비스가 다운되어 데이터가 손실 되는 것을 방지하기 위해 분리했다.

Airflow 현재 배포 구성 https://gitlab.euso.kr/data_visualization/lime_flow



- Custom airflow image

- Redis:latest image
- postgres:13 image

- Custom airflow image

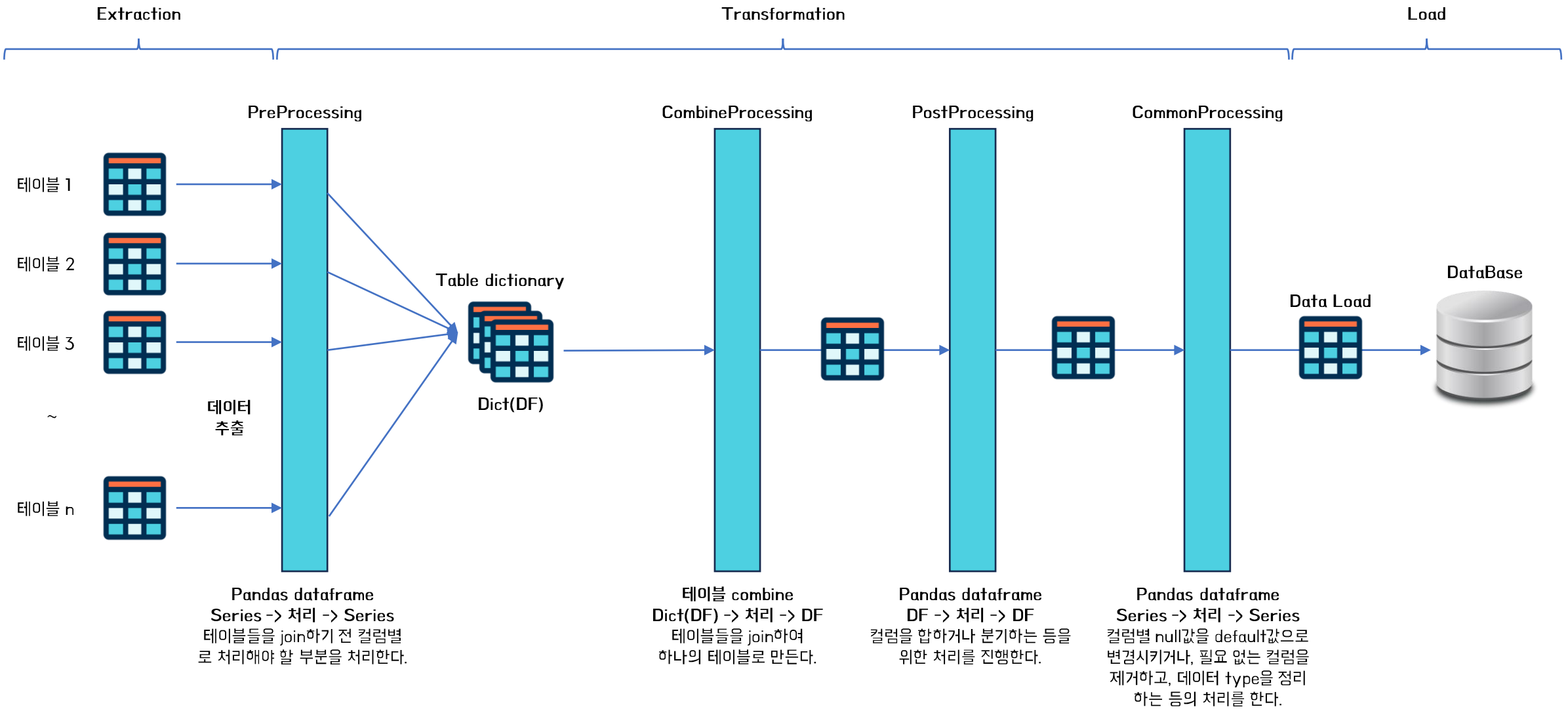
Custom Operator 구성 -- KistepDMOperator

- 기존 kistep, Tipa 프로젝트에서 data생성 코드의 문제점
  - 모놀리식 구조로 이루어져 있어 개발 및 유지보수의 어려움
  - 각 컬럼에 대한 데이터 처리 내용을 파악하기 힘듦
  - Default 값이 따로 없으며, 데이터 타입을 한번에 명시하지 않아 type 때문에 문제가 생길 가능성이 있음
  - Try~ Exception~ 구문만으로 에러처리가 되어있어 retry 등에 대한 처리가 안되어 있음
  - Worker 개념이 없이 하나의 프로세스로만 처리하여 여러 작업을 동시에 처리하지 못함
  - Print로만 Log처리가 되어 있어 문제가 생겼을 때 정확한 파악을 하기 어려움
  - 에러가 생긴 부분에 대한 재처리가 힘들다.

- Kistep Data mart 전용 Custom Operator을 작성한 이유
  - 기존 모놀리식 구조를 분리, 각 컴포넌트 별로 역할을 부여하여 개발 및 유지보수 속도를 높임
  - 데이터 처리에 대한 맵핑 dictionary를 사용하여 각 컬럼에 대한 데이터 처리 내용을 한눈에 파악할 수 있음
  - Airflow에서 제공해주는 log 객체를 이어받아 각 처리내용을 log파일로 남겨서 web ui로 해당 에러 부분을 디버깅 할 수 있음
  - Celery Worker를 사용하여 동시에 여러 작업을 처리할 수 있음(Worker 수와 task 수에 따라 다름)
  - 데이터 처리 내용을 dag파일과 분리하여 dag 파일의 스케줄링 및 의존성 파악을 쉽도록 함
  - 문제가 발생할 경우 dag에 명시한 retry로 여러 번 시도가 가능하며, web ui에서 에러 내용을 파악하고 해당 내용만 다시 돌리는 것도 가능함

Custom Operator 구성 -- KistepDMOperator

- Data 전처리 플로우



Custom Operator 구성 -- KistepDMOperator

Custom

Constants

Processors

BaseProcessor.py

BaseProcessor 클래스
아래 모든 프로세서들이 이 프로세서를 상속하며, 각 프로세서들에서 공통으로 사용할 수 있는 함수를 정의

PreProcessor.py

PreProcessor 클래스
데이터를 추출한 직후 combin하기 전 컬럼별로 처리해야 할 내용을 정의한다. 컬럼별로 함수를 만든다.
Input : pandas series
Output : pandas series

PostProcessor.py

PreProcessor 클래스
필요한 컬럼을 합하거나 제거하는 등 테이블 전체를 바라봐야 하는 작업을 진행한다.
Input : pandas DataFrame
Output : pandas DataFrame

CommonProcessor.py

CommonProcessor 클래스
컬럼별 default 값, null처리, type등을 맞춘다.

Input : pandas DataFrame
Output : pandas DataFrame

Data Mart Specification.py

DataMartSpecification dictionary를 담고있는 클래스

Datamart를 구성하는 컬럼, 원본 소스가 있는 테이블, 전처리 메서드(PreProcessing, PostProcessing, CommonProcssing) 맵핑 정보, 컬럼별 default값, 컬럼별 type정보 등을 담고있는 Datamart 정의 딕셔너리

(뒤에 자세히 기술)

Specification Parser.py

SpecificationParser 클래스

DataMart 맵핑 정보를 Opertator에서 원하는 형태로 변형시켜주는 함수 집합

KistepDMOperator.py

Dag에서 날짜값 등과 함께 요청을 받아 각 Processor들을 실제로 구동시키는 부분

데이터를 추출, 처리, 로드 하는 로직이 담겨있다.

데이터를 추출하는 메서드는 Operator에서만 작성한다. 이는 각 모듈별로 역할을 분명히 하기 위함이다.

MariaHook.py

MariaDB와 연결을 관리하고 데이터를 추출하고 삽입하기 위한 코드가 작성되어 있다.

DAG.py

파이프라인을 위한 스케줄을 관리하고 각 task간의 종속성을 관리한다.

Custom Operator 구성 -- KistepDMOperator

- BaseProcessor

```
class BaseProcessor:
    def __init__(self) -> None:
        self.space_pattern = re.compile(r"\n|(\\r)|(\\n)|\r",
re.UNICODE)
        self.space_pattern2 = re.compile(r"\s+|['&#10;']+",
re.UNICODE)
        self.space_pattern3 = re.compile(r"\s+", re.UNICODE)
        self.special_pattern =
re.compile(r"[◆■○●◎◑☞★※]+|※|[\*]+|[▶▲]|[《》‹›◇
\[\]]|[-_\|]|[-+]|[「」]|[! , … • · ≫\'`‘“”:;?]")
        self.multi_comma_patter =re.compile(r"[.]{2,}")
        self.uniqu_number_pattern =
re.compile(r"[ⅠⅡⅢⅣⅤⅥⅦⅧⅨⅩ]|[ⅰⅱⅲⅳⅴⅵⅶⅷⅸⅹ]|[①②③④
⑤⑥⑦⑧⑨⑩]")
        self.quotation_pattern = re.compile(r"[\"\']")

    # 한글 유무 확인
    def is_kor(self, string):
        '''
        한글이 있는 경우 : True
        한글이 없는 경우 : False
        '''
        hangul = re.compile('[ㄱ-ㅎㅣ가-힣]+')
        if string:
            result = hangul.findall(string)
        else:
            result = ""
        if len(result) == 0:
            return False
        else :
            return True

    # 공백 제거
    def remove_space(self, string):
        regex_text = self.space_pattern.sub(" ", string)
        regex_text = self.space_pattern3.sub(" ", regex_text)
        regex_text = regex_text.strip()
        return regex_text

    # 숫자 컬럼 공백제거
    def remove_space_num_col(self, string):
        regex_text = self.space_pattern.sub(" ", string)
        regex_text = self.space_pattern3.sub(" ", regex_text)
        regex_text = regex_text.strip()
        return regex_text

    # 특수문자 제거
    def remove_special(self, string):
        regex_text = self.special_pattern.sub(" ", string)
        regex_text = self.multi_comma_patter.sub(".",
regex_text)
        regex_text = self.space_pattern.sub(" ", regex_text)
        regex_text = self.space_pattern3.sub(" ", regex_text)
        regex_text = regex_text.strip()
        return regex_text
~
```

- PreProcessor

```
from .BaseProcessor import BaseProcessor


class PreProcessor(BaseProcessor):
    def __init__(self, pj_hist_mapper={}, cd_dtl_mapper={}) ->
None:
        super().__init__()
        self._pj_hist_mapper = pj_hist_mapper
        self._cd_dtl_mapper = cd_dtl_mapper


    # 이전과제 고유번호
    def make_pre_pre_pjt_id(self, pjt_id_series):
        res = pjt_id_series.map(lambda x: self._pj_hist_mapper[x]
if x in self._pj_hist_mapper.keys() else "")
        return res

    # 국문 과제명
    def make_pre_kor_pjt_nm(self,pjt_nm_series):
        res = pjt_nm_series.map(lambda x: x if self.is_kor(x) else
"")
        return res

    # 영문 과제명
    def make_pre_eng_pjt_nm(self,pjt_nm_series):
        res = pjt_nm_series.map(lambda x: x if not (self.is_kor(x))
else "")
        return res

    # 적용분야분류1코드 처리 함수
    def make_pre_appl_area_cls_cd(self,appl_area_cls_cd_series):
        res = appl_area_cls_cd_series.fillna("999999").map(lambda
x: x[3:] if (x != "999999") and (len(x) == 6) else "")
        return res

    # 적용분야분류1코드 명 처리 함수
    def make_pre_appl_area_cls_nm(self,appl_area_cls_cd_series):
        res = appl_area_cls_cd_series.fillna("999999").map(lambda
x : self._cd_dtl_mapper[x] if (x != "999999") and (x in
self._cd_dtl_mapper.keys()) else "")
        return res

    # 연구분야 분류 1코드
    def make_pre_rsch_area_cls_cd(self,rsch_area_cls_cd_series):
        res = rsch_area_cls_cd_series \
                .fillna("UK9999") \
                .map(lambda x: x[3:] if (x != "UK9999") and (len(x)
== 9) else "UK9999")
        return res

    # 연구분야 분류 1 코드명
    def make_pre_rsch_area_cls_nm(self,rsch_area_cls_cd_series):
        res = rsch_area_cls_cd_series \
                .fillna("분류체계없음") \
                .map(lambda x : self._cd_dtl_mapper[x] if
(len(x) > 0) and (x in self._cd_dtl_mapper.keys()) else "분류체계
없음")
        return res
~
```

- PostProcessor

```
from .BaseProcessor import BaseProcessor

class PostProcessor(BaseProcessor):
    def __init__(self, offc_cd_mapper={}) -> None:
        super().__init__()
        self._offc_cd_mapper = offc_cd_mapper

    # 분석용 컬럼 데이터 생성
    def make_analysis_target_text(self, combined_df):
        res_df = combined_df.copy()
        analysis_target_cols = ['rsch_goal_abstract',
'exp_efct_abstract', 'rsch_abstract', 'kor_kywd', 'eng_kywd']

        res_df[analysis_target_cols] =
res_df[analysis_target_cols].fillna('')

        # 대상 텍스트 모으기
        res_df["analysis_target_text"] = res_df.apply(lambda row:
' '.join([row['rsch_goal_abstract'], row['exp_efct_abstract'],
row['rsch_abstract'], row['kor_kywd'], row['eng_kywd']]), axis=1)

        # 특수 문자 제거
        res_df["analysis_target_text"] =
res_df["analysis_target_text"].map(lambda string :
self.remove_special(string))

        # 특수 숫자 제거
        res_df["analysis_target_text"] =
res_df["analysis_target_text"].map(lambda string :
self.remove_special_num(string))
        return res_df

    # 부처 코드 병합하기
    def make_prog_mstr_cd(self, combined_df):
        res_df = combined_df.copy()
        res_df["prog_mstr_cd"] = res_df.apply(lambda row:
row["prog_mstr_cd_NEW"] if row["prog_mstr_cd_NEW"] else
row["prog_mstr_cd_OLD"], axis=1)
        return res_df

    # 부처 코드 명 생성
    def make_prog_mstr_nm(self, combined_df):
        res_df = combined_df.copy()
        res_df["prog_mstr_nm"] =
res_df["prog_mstr_cd"].map(self._offc_cd_mapper)
        return res_df

    # pjt_id 없는 것 확인해서 drop 하기
    def make_doc_id(self, combined_df):
        res_df = combined_df.copy()
        res_df = res_df.dropna(subset=["doc_id"], axis=0)
        return res_df
~
```

- CommonProcessor

```
from .BaseProcessor import BaseProcessor

class CommProcessor(BaseProcessor):
    def __init__(self) -> None:
        super().__init__()

    def l_r_strip(self, series):
        res = series.map(lambda x: x.strip() if x else x)
        return res

    def trash_space_to_one_space(self, series):
        res = series.map(lambda x: self.remove_space(x) if x else
x)
        return res
```

Custom Operator 구성 -- KistepDMOperator

- Data Mart Specification

```python
from .Processors.PreProcessor import PreProcessor
from .Processors.PostProcessor import PostProcessor
from .Processors.CommonProcessor import CommProcessor

class DocumentNtisSpecification:
    def __init__(self, pj_hist_mapper={}, cd_dtl_mapper={}, offc_cd_mapper={}) -> None:
        self.pre = PreProcessor(pj_hist_mapper=pj_hist_mapper, cd_dtl_mapper=cd_dtl_mapper)
        self.post = PostProcessor(offc_cd_mapper=offc_cd_mapper)
        self.comm = CommProcessor()
        self.DOCUMENT_SPECIFICATION = {
            "mappings" : {
                ~
                "pjt_prfrm_org_nm": {# <- DataMart 컬럼명
                    "comment" : "과제수행기관명", # <- DataMart 컬럼 코멘트
                    "source" : [
                        {
                            "tbl" : "TIA_PJ_IRIS", # <- 원본 소스 테이블 명
                            "col" : "MAIN_RSCH_ORG_NM" # <- 원본 소스 컬럼명
                        }
                    ],
                    "type": "str", # <- DataMart 컬럼의 기본 타입
                    "default_value": "", # <- DataMart 컬럼의 기본 값
                    "pre_processing" : [self.pre.make_pre_pjt_prfrm_org_nm],
                    "post_processing" : [self.post.make_prog_mstr_cd, self.post.make_prog_mstr_nm],
                    "common_processing": [self.comm.l_r_strip, self.comm.trash_space_to_one_space]
                },
                ~

            }
            "combine_settings" : [
                {
                    "mode": "merge",
                    "how": "left",
                    "left": "TIA_PJ_IRIS",
                    "right": "TIA_INV_ANA_EVAL_TRGT_BZ_IRIS",
                    "on": ["org_bdgt_prog_cd", "stan_yr"]
                },
                ~
            ]
            "drop_target_cols" : ["prog_mstr_cd_NEW", "prog_mstr_cd_OLD"]
        }

    def get_document_specification(self):
        return self.DOCUMENT_SPECIFICATION
```

Pre, Post, Common 프로세서의 객체를 생성한다.

"mappings" : DataMart의 컬럼별 기본 정보, 처리 내역 맵핑 dictionary
"pre_processing", "post_processing", "common_processing"에는 각각 위에서 선언한 프로세서 객체의 매서드를 리스트로 나열한다. 먼저 오는 매서드 부터 처리된다.

여러 테이블을 합쳐서 하나의 datamart로 만들 때 필요한 내용을 정의한다. Argument들은 pandas의 merge메서드, concat 메서드와 동일하다.

전처리 과정에서는 필요했지만 최종적으로 제거해야 할 컬럼목록을 리스트로 나열한다.

Custom Operator 구성 -- KistepDMOperator

- MariaHook

```python
import logging
from sqlalchemy import create_engine
from sqlalchemy.orm import Session
from sqlalchemy.exc import SQLAlchemyError, IntegrityError
from airflow.hooks.base import BaseHook


import pandas as pd

class MariaHook(BaseHook):
    MARIA_DEFAULT_PORT = 3306

    def __init__(self, conn_id, pool_size=10, pool_recycle=500, max_overflow=10):
        super().__init__()
        self._conn_id = conn_id
        self._pool_size = pool_size
        self._pool_recycle = pool_recycle
        self._max_overflow = max_overflow
        self._session = None
        self.logger = logging.getLogger(__name__)

    def __enter__(self):
        return self

    def __exit__(self, exc_type, exc_val, exc_tbl):
        self._session.commit()
        self.close()

    def get_session(self):
        config = self.get_connection(self._conn_id)

        if (not config.host):
            raise ValueError(f"No host specified in connection {self._conn_id}")
        host = config.host
        port = config.port or self.MARIA_DEFAULT_PORT
        id = config.login
        pwd = config.password
        schema = config.schema

        engine = create_engine(f"mysql+pymysql://{id}:{pwd}@{host}:{port}/{schema}?charset=utf8mb4", pool_size=self._pool_size,
pool_recycle=self._pool_recycle, max_overflow=self._max_overflow)
        self._session = Session(engine)
        self.logger.info(f"Get session by conn_id= {self._conn_id}")
        return self._session

    def get_conn(self):
        if self._session is None:
            self._session = self.get_session()
        self._conn = self._session.connection()
        return self._conn

    def close(self):
        if self._session:
            self._session.commit()
            self._session.close()
        self._session = None
```

```python
    def _read_sql(self, query):
        conn = self.get_conn()
        try:
            self.logger.info(f"Try to read df by | {query}")
            df = pd.read_sql(query, conn)
        except SQLAlchemyError as err:
            self._session.rollback()
            self.logger.info(err)
            raise SQLAlchemyError(err)
        except Exception as err:
            self._session.rollback()
            self.logger.info(err)
            raise Exception(err)
        finally:
            self.close()
        return df

    def _to_sql(self, df, table_name=None, if_exists='append', index=False) -> None:
        if table_name is None:
            raise ValueError(f"table_name is not defined (table_name={table_name})")

        conn = self.get_conn()
        try:
            df.to_sql(table_name, con=conn, if_exists=if_exists, index=index)
        except IntegrityError as err:
            self._session.rollback()
            raise IntegrityError(err)
        except SQLAlchemyError as err:
            self._session.rollback()
            raise SQLAlchemyError(err)
        except Exception as err:
            self._session.rollback()
            raise Exception(err)
        finally:
            self.close()
        return None
```

Custom Operator 구성 -- KistepDMOperator

- KistepDMOperator

```python
from .constants.document import DocumentNtisSpecification
from .constants.Parser import SpecificationParser

from airflow.models import BaseOperator
from airflow.utils.decorators import apply_defaults

from .MariaHook import MariaHook

import logging
import pandas as pd

class KistepDMOperator(BaseOperator):
    @apply_defaults
    def __init__(self, from_conn_id, to_conn_id, target_dm=None, insert_target_table_nm=None, **kwargs):
        super().__init__(**kwargs)
        self.logger = logging.getLogger(__name__)
        self._from_conn_id = from_conn_id
        self._to_conn_id = to_conn_id
        self.from_hook = MariaHook(self._from_conn_id)
        self.to_hook = MariaHook(self._to_conn_id)
        self._target_dm = target_dm
        self._insert_target_table_nm = insert_target_table_nm

        self.DOCUMENT_SPECIFICATION = None
        self.parser = self.get_parser(self._target_dm)



    def get_parser(self, target_dm):
        if target_dm == "document_ntis":
            offc_cd_mapper = self.get_offc_cd_nm_mapper("TBU_OFFC_CD_IRIS")
            pj_hist_mapper = self.get_pj_hist_mapper("TIA_PJ_HIST_IRIS")
            commcd_mapper = self.get_commcd_nm_mapper("TCO_CD_DTL_IRIS")
            self.DOCUMENT_SPECIFICATION = DocumentNtisSpecification(pj_hist_mapper=pj_hist_mapper,
                                                                    cd_dtl_mapper=commcd_mapper,
                                                                    offc_cd_mapper=offc_cd_mapper
                                                                    ).get_document_specification()

            parser = SpecificationParser(self.DOCUMENT_SPECIFICATION)
        elif target_dm == "document_iris":
            pass
        elif target_dm == "player":
            pass
        else:
            raise ValueError
        return parser


        ~
```

```python
    # pre-processing
    def get_base_df_dic(self, specification_parser):

        ~

        return base_df_dic

    # combined_df 생성
    def get_combined_base_df(self, specification_parser, base_df_dic):

        ~

        return combined_base_df

    # post-processing
    def get_post_processed_df(self, specification_parser, combined_df):

        ~

        return combined_df

    # common-processing
    def get_common_processed_df(self, specification_parser, post_processed_df):

        ~

        return post_processed_df

    def insert_to_db(self, df, table_name, chunk_size=10000):
        num_chunks = (len(df) // chunk_size) + 1

        for num in range(num_chunks):
            start_idx = num * chunk_size
            end_idx = start_idx + chunk_size
            chunk_df = df[start_idx:end_idx]
            self.to_hook._to_sql(chunk_df, table_name=table_name)

    def execute(self, context, *args, **kwargs):
        self.logger.info(f"context : '{context}'")
        self.logger.info(f"context : '{args}'")
        self.logger.info(f"context : '{kwargs}'")

        parser = self.parser
        # pre-processing
        base_df_dic = self.get_base_df_dic(parser)

        # combine-processing
        combined_df = self.get_combined_base_df(parser, base_df_dic)

        # post-processing
        post_processed_df = self.get_post_processed_df(parser, combined_df)

        # common-processing
        common_processed_df = self.get_common_processed_df(parser, post_processed_df)

        # load to db
        self.insert_to_db(common_processed_df, self._insert_target_table_nm, chunk_size=10000)
```