Airflow Udemy(The Complete Hands-On Introduction to Apache Airflow)

<에어플로우?>

- 데이터 투 두 리스트라고 생각하면 됨. 추출, 처리, 로딩 한 눈에 볼 수 있음
- 에어플로우 파이썬으로 사용 가능.
- 데이터 모니터링 가능
- 데이터 가지고 잡다한 일 자동화 할 수 있고, 확장성도 좋음

<구성요소>

1. 웹 서버: 뷰, 매니지, 모니터 할 수 있음. 예) 차의 대시보드
2. 스케줄러: 어떤 태스크가 언제 실행되어야하는지 담당
3. 메타 데이터베이스: 에어플로우의 메모리라고 생각하면 됨. 내 태스크나 상태를 저장하는 데이터베이스
4. 트리거러: deferrable(연기할 수 있는) 태스크들(외부 요인 기다리는 태스크) 담당
5. executor- 어떻게 태스크가 실행될건지 결정 예) 교통정리 담당
6. queue - 실행되기를 기다리는 태스크의 리스트
7. worker - 실제로 태스크를 실행하는 과정들

<핵심 개념>

1. DAG(Directed Acyclic Graph): 의존성을 반영하는 방식으로 내가 실행하고자 하는 태스크의 컬렉션. 예) 요리 레시피 + loop이 있으면 DAG가 아님 ! Acyclic → 어떠한 사이클도 없어야함.
2. Operator: DAG 안에 있는 하나의, idempotent(멱등성)한 태스크, 워크플로우를 하나의 관리 가능한 태스크로 나눔 예) 레시피에서 'break 5 eggs'
3. task/task instance: 태스크는 오퍼레이터의 특정한 인스턴스. 오퍼레이터가 dag에 명시되면, 태스크가 됨. 예시) '7월 1일 2시에 break 5 eggs' 는 task instance.
4. workflow: DAG에 정의된 모든 태스크와 의존성을 포함한 전체 프로세스.

<언제 써야하냐?>

쓰면 안됨 > 데이터 처리 도구 아님, 실시간 아니고 배치 스케줄링기반임, 저장 시스템도 아님,

쓰면 좋음 > 고빈도, sub-minute 스케줄링, 빅데이터셋을 '직접' 프로세싱 할 때. (**프로세싱 하는 것이 아니라 '오케스트레이팅'할 때 사용됨**), 실시간 데이터 스트리밍(실시간으로 도착하는 데이터를 배치 스케줄

링 형태로 오케스트레이팅 할 수 있다는 뜻임), 간단한 의존성과 간단한 선형 워크플로우 가지고 있는 경우

<에어플로우 운영 위한 다른 아키텍쳐들>

- single node architecture: 에어플로우의 모든 요소가 한 머신에서 실행될 때
  - 메타 데이터 베이스는 아키텍처 내부의 모든 요소가 얘를 통해 소통함
  - 작은 워크플로우 운용하기 가능, 설치하기에 간단
- multi node architecture: 여러 컴퓨터나 서버를 통해 실행되는 경우
  - 노드 A에는 서버만 있고, 노드 B에는 스케줄러 있고, DEF에는 worker들 있고 등

<도커가 뭐냐>

- 도커: 패키지랑 소프트웨어 돕기위한 툴
- 장점
  - 패키징: 컨테이너라고 불리는 곳에 앱,코드, 라이브러리 저장
  - 일관성: 어떤 컴퓨터에서도 같은 컨테이너는 똑같이 작동함
  - 고립성(isloation): 한 컨테이너는 다른 컨테이너와 구분됨
  - 효율성: 가벼운 소프트웨어
  - 사용하기 쉬움
- 도커에서는 만약 내 웹사이트에 여러 파이썬 라이브러리나 데이터베이스 필요할 때 한 컨테이너에서 잘 작동할 수 있게 함. 좋은 패키징 시스템임

section4. coding your first data pipeline with airflow

<DAG>

- Directed Acyclic Graph: 에어플로우에서 워크플로우를 정의하는 구조
  - directed: 작업(task)가 특정 순서대로 실행되어야 함
  - Acyclic: 루프가 없어야함

  → DAG는 여러 task 간의 의존성을 설정해서 실행 순서를 조정함

DAG 주요 키워드들

<operator>

- DAG에서 태스크 정의하는 기본 단위. 특정 작업(bash 명령 실행이나 python 코드 실행) 수행
  - BashOperator: 쉘 명령 실행
  - PythonOperator: Python 함수 실행
  - DummyOperator: 아무 작업도 하지 않는 placeholder
  - BranchPythonOperator: 조건에 따라 다른 태스크로 분기
  - Sensor: 특정 조건이 충족될 때 까지 대기

- ‘하나의 태스크’가 키워드라서, 파이썬 오퍼레이터에서 데이터 클렌징이랑 데이터 처리는 각각의 오퍼레이터에 넣어야함.
- 종류

1. action operator: 액션 실행
2. transfer operator: 데이터를 transfer
3. sensors: 조건이 맞을 때까지 기다림

<provider>

- 다양한 외부 시스템과의 연결을 가능하게 하는 플러그인 모듈
- 만약에 snowflake, aws 등 다양한 provider랑 연결하고 싶으면, pip 이용해 설치하기만 하면 에어플로우 내에서 이을 수 있음
- 패키지: apache-airflow-providers-google

<sensors>

- 특정 조건이 만족될 때까지 DAG 실행 대기시키는 특수한 Operator
- poke_interval과 timeout이라는 파라미터 알아야함
  - poke_interval: 센서가 주기적으로 조건을 확인하는 간격 설정
  - timeout: 센서가 조건을 만족할 때까지 최대 대기 시간을 설정

<hook>

- 외부 시스템과 상호작용하기 위해나 API 또는 연결을 처리하는 유틸리티. 내가 외부 도구/서비스랑 인터랙트 해야할 때는, ‘hook’ 기억
- dependency 표시는 항상 모든 프로세스 마지막에 해주기.

Q. provider도 외부 시스템과의 연결 아닌가? provider와 hook 둘이 뭐가 다른거지 → hook의 경우 데이터를 가져오거나 작업을 수행하는 ‘저수준 API’ 제공. 즉, 외부 시스템과의 연결을 다루는 개별적인 기능을 제공, Provider는 외부 시스템과의 통합을 보다 쉽게 할 수 있도록 다양한 컴포넌트를 패키징한 ‘고수준’라이브러리. 즉, 여러 개의 Hook, Operator 등을 하나의 패키지로 제공하여 사용자가 외부 시스템과 통합할 때 더 많은 기능 제공

<dataset>

- 특정 데이터셋에 의존하는 DAG 간의 관계를 설정하는 새로운 방식. 에어플로우에서 데이터 흐름을 추적하고 관리하는 객체. 주로 태스크 간에 데이터 의존성 관리하기 위해서 사용됨
- 데이터셋 기반으로 DAG 트리거 설정 가능
- URI: 데이터셋을 고유하게 식별하는 경로 또는 식별자. 일반적으로 파일 경로, 데이터베이스 테이블, 데이터 엔드포인트를 URI로 저장 가능
- EXTRA: 추가적인 메타데이터를 저장할 수 있는 필드. 예를 들면, 데이터셋 버전, 포맷, 추가 설명 저장 가능

```
from airflow import Dataset
from airflow.decorators import task

my_dataset = Dataset('/path/to/dataset')

@task(outlets=[my_dataset])
def produce_data():
    # 데이터 생성 작업

@task(inlets=[my_dataset])
def consume_data():
    # 데이터 사용 작업
```

- 에어플로우에서 @task 데코레이터는 특정 함수에서 에어플로우에서 실행 가능한 '태스크'로 변환함. 즉, 스케줄러가 해당 태스크를 이제 관리함
- `produce_data` 함수는 데이터 생성하는 작업하고, 생성된 데이터가 my_dataset에 기록됨 → `outlets` 사용해서 데이터 외부로 전달함.
- `consume_data` 함수는 my_dataset에 결과를 기록하고, `inlets`는 이 태스크가 의존하는 데이터셋임

<Executor>

- DAG 태스크를 실행하는 방식 및 백엔드 환경 정의
  - SequentialExecutor: 태스크 하나씩 실행(기본 설정)
  - LocalExecutor: 멀티프로세싱 사용해서 태스크 병렬 실행
    - '로컬머신'에서 여러 태스크를 병렬로 실행하는 방식
    - 작은 규모의 데이터와 작업에 적합
  - CeleryExecutor: 분산 환경에서 태스크 실행
    - 여러 서버에서 태스크를 병렬로 실행할 수 있으며, 작업 큐를 통해 분산된 시스템에서 태스크 처리
  - KubernetesExecutor: 쿠버네티스 기반 클러스터에서 태스크 실행
- local executor는 SQL 안 쓰면 여러 태스크 한 번에 처리할 수 있음
- celery executor;
- result backend: 샐러리와 브로커에 의해서 저장된 태스크의 상태가 있는 곳

<adios repetitive patterns>

- 반복 작업을 제거하거나 간소화하는 접근 방식
- 에어플로우에서는 반복적인 DAG 구조를 효율적으로 관리하기 위해서 SubDAG나 TaskGroup을 사용함(SubDAG: DAG 내부에 또 다른 DAG 정의, Task Group: 태스크를 그룹으로 묶어 반복 작업을

정리)

<지금까지 코드리뷰>

```
from airflow import DAG
from airflow.providers.postgres.operators.postgres import PostgresOpera
from datetime import datetime
from airflow.providers.http.sensors.http import HttpSensor # type: ignc
from airflow.providers.http.operators.http import SimpleHttpOperator #
from airflow.operators.python import PythonOperator
from airflow.providers.postgres.hooks.postgres import PostgresHook

import json
from pandas import json_normalize


def _process_user(ti):
    user = ti.xcom_pull(task_ids="extract_user")
    user = user['result'][0]
    processed_user = json_normalize({
        'firstname': user['name']['first'],
        'lastname': user['name']['last'],
        'country': user['location']['country'],
        'username': user['login']['username'],
        'password': user['login']['password'],
        'email': user['email'] })
    processed_user.to_csv('/tmp/processed_user.csv', index = None, head

def _store_user():
    hook = PostgresHook(postgres_conn_id='postgres')
    hook.copy_expert(
        sql="COPY users FROM stdin WITH DELIMITER as','",
        filename='/tmp/processed_user.csv')


with DAG('user_processing', start_date=datetime(2022, 1, 1),
         schedule_interval='@daily', catchup=False) as dag:

    create_table = PostgresOperator(
        task_id='create_table',
        postgres_conn_id='postgres',
        sql='''
            CREATE TABLE IF NOT EXISTS users (
                firstname TEXT NOT NULL,
                lastname TEXT NOT NULL,
```

```
                country TEXT NOT NULL,
                username TEXT NOT NULL,
                password TEXT NOT NULL,
                email TEXT NOT NULL
            );
        '''
    )

    is_api_available = HttpSensor(
        task_id='is_api_available',
        http_conn_id='user_api',
        endpoint='api/')


    extract_user = SimpleHttpOperator(
        task_id = 'extract_user',
        http_conn_id='user_api',
        endpoint='api/',
        method='GET',
        response_filter = lambda response: json.loads(response.text),
        log_response = True
    )


    process_user = PythonOperator(
        task_id = 'process_user',
        python_callable = _process_user
         )


    store_user = PythonOperator(
        task_id='store_user',
        python_callable=_store_user
        )

    create_table >> is_api_available >> extract_user >> process_user >>
```

- 왜 `process_user`와 `store_user`는 `with DAG`에 정의한 것 말고도 `def`로 함수를 정의해야 하는가? → 이 두 태스크는 Pythonoperator를 사용하여 실행되는 함수이므로, 에어플로우에서 python 함수는 python_coallable 인수로 넘겨져야함. `PythonOperator` 는 실제로 실행할 python함수를 참조해야하니까, 해당 애들을 def로 정의하는 것

```
from asyncio import Task
from airflow import DAG, Dataset
from airflow.decorators import task

from datetime import datetime

my_file = Dataset("/tmp/my_file.txt")
my_file_2 = Dataset("/tmp/my_file_2.txt")

with DAG(
    dag_id="producer",
    schedule="@daily",
    start_date=datetime(2022,1,1),
    catchup=False
):


    @task
    def update_dataset(outlets=[my_file]):
        with open(my_file.uri, "a+") as f:
            f.write("producer update")

    @task
    def update_dataset_2(outlets=[my_file_2]):
        with open(my_file_2.uri, "a+") as f:
            f.write("producer update")

    update_dataset() >> update_dataset_2()
```

- dataset(데이터를 생성하거나 처리하는 작업의 입출력 관리하는 데 사용) 사용하여 두 개의 파일을 업데이트하는 producer 작업을 정의한 것.
- `outlet` 역할: 태스크가 데이터를 출력하는 위치
- 파이썬에서 with: 블록 안에서 리소스 열고, 블록 벗어나면 자동으로 해당 리소스 닫거나 정리해주는 역할. open과 close를 따로 지정해줄 필요 없음

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

from datetime import datetime

with DAG('parallel_dag', start_date=datetime(2022, 1, 1),
```

```
    schedule_interval='@daily', catchup=False) as dag:

    extract_a = BashOperator(
        task_id='extract_a',
        bash_command='sleep 1'
    )

    extract_b = BashOperator(
        task_id='extract_b',
        bash_command='sleep 1'
    )

    load_a = BashOperator(
        task_id='load_a',
        bash_command='sleep 1'
    )

    load_b = BashOperator(
        task_id='load_b',
        bash_command='sleep 1'
    )

    transform = BashOperator(
        task_id='transform',
        queue='high_cpu',
        bash_command='sleep 30'
    )

    extract_a >> load_a
    extract_b >> load_b
    [load_a, load_b] >> transform
```

→ 이 코드는 에어플로우 DAG를 사용해서 병렬 처리와 의존성 관리를 구성하려는 예시.

- extract a,b는 데이터 추출, load a,b는 로드, transform은 데이터 변환하는 태스크, 30초동안 대기하다가, high_cpu에 할당됨. 시스템 리소스를 더 많이 사용하는 태스크로 설정 됨
- Q. 왜 여기서 BashOperator를 써야하는가?: BashOperator는 Bash 쉘 명령어를 실행하는 데 특화되어 있음. 시스템 명령어(curl, psql,wget 등) 스크립트 실행할 때 사용
    - Q. hook은 안됨?: 외부 시스템과의 상호작용 있을 때(e.g., 데이터베이스 연결이나 외부 API와의 복잡한 상호작용 처리할 때)
- Xcom: task 간에 '작은' 규모의 '메타'데이터를 공유해야할 때 (데이터 자체를 주고받는 것은 아님에 주의)
- airflow에서 branch: 조건에 따라 특정 경로를 선택적으로 실행하도록 하는 기능

- trigger rule: 특정 태스크가 실행되기 전에 어떤 조건을 만족해야하는지 정의하는 규칙
    - all_success(기본값): 모든 부모태스크가 성공적으로 완료되어야 현재 태스크 실행됨
    - all_failed: 부모 태스크들이 모두 실패해야 현재 태스크가 실행됨
    - one_success: 하나라도 성공해야
    - one_failed: 하나라도 실패해야
    - none_failed: 실패한 것이 없어야
    - none_skipped: 스킵되지 않아야
    - dummy: 태스크가 특정 조건을 만족하지 않으면 무시하는 조건으로, 주로 테스트나 디버깅에서 사용
- elastic search: 분산형 검색 및 분석 엔진. 대규모 데이터에 대해 빠른 검색과 분석 가능하게 함. 주로 로그 처리나 메타 데이터 추적 사용함.
- plugin 어떻게 작동하는지?: 기본 기능의 확장 가능. 에어플로우의 특정 기능을 추가하거나, 사용자 정의 오퍼레이터, 센서, 훅 만들어서 워크플로우에서 사용 가능하게 함

<참고한 링크>

- https://yozm.wishket.com/magazine/detail/2874/
- https://www.bucketplace.com/post/2021-04-13-%EB%B2%84%ED%82%B7%ED%94%8C%EB%A0%88%EC%9D%B4%EC%8A%A4-airflow-%EB%8F%84%EC%9E%85%EA%B8%B0/
    - 쿠버네티스에 대한 이해가 부족해서 글 이해하기 힘들었음
        - 쿠버네티스: 컴퓨터에서 작업을 실행하고 관리하는 '감독관' 같은 것. 여러대의 컴퓨터(가상 컴퓨터도 포함)을 관리하며 어떤 작업을 어디에 할당할지, 어떻게 실행할지 결정
        - 쿠버네티스는 pod라는 작은 상자 만들어서 작업을 그 안에 넣어서 실행함
            - 상자: 필요한 프로그램과 설정을 담고 있는 일회용 작업 공간
        - 쿠버네티스 익스큐터: airflow에서 '이 task 실행해야함'이라는 신호가 오면, 쿠버네티스 익스큐터가 pod라는 작은 상자 만들고, 그 안에서 task를 실행함. task가 끝나면 상자는 사라짐
        - 장점: task 실행할 때마다 새 상자 만들어서 실행하니까 다른 작업에 영향 주지 않음 + task 끝나면 상자 사라지니까 컴퓨터 자원 계속 차지하지 않음 + 워커를 직접 관리할 필요가 없음. 쿠버네티스가 알아서 처리해주니까.
        - 단점: 상자 만드는데 시간이 좀 걸림.. 만약 일처리하는데 1초, 상자만드는데 5초면 무슨 의미? + 설정이 복잡하고, 비교적 최근에 나온 개념이라 자료가 많지 않아 어려움

        즉, 쿠버네티스는 학교의 '배식로봇'같은 역할. 학생들(=task)이 밥을 요청하면, 쿠버네티스가 작은 도시락 상자(=pod) 만들어서 밥 담아줌. 학생이 밥 다 먹으면 도시락 상자는 사라짐. 음식을 효율적으로 나눠줄 수는 있지만, 도시락 상자 만드는데 시간 걸리고 로봇을 세팅하는게 어렵다는 단점이 있음