## Chapter 1 : Apache Airflow 살펴보기

### #(일반)데이터 파이프라인 실행

- 여러 태스크(= 동작)으로 구성

Ex) 실시간 날씨 대시보드 구현

(1) 날씨 API 이용, 예보 데이터 수집 – 태스크(A)
(2) 데이터 가공 – 태스크(B)
(3) 데이터 대시보드로 전송 – 태스크(C)

- 태스크 간 의존성(비순환)
  : 태스크(A) 완료 되야만, (B) 태스크 실행됨.

### #Airflow 그래프 실행

- DAG : 파이프라인 기본구조
- DAG의 비순환 종속성
  : DAG(태스크) 실행 -> 대기열 추가 -> DAG(태스크) 실행 .... 형태로 프로세스 진행
  - ➢ 실행 미완료(= 의존성 미해결) 시, 대기열 추가 (X)
  - ➢ 더 이상 실행할 작업 없을 시, 전체 파이프라인 실행 완료
- DAG(태스크) 병렬처리 가능 -> 실행시간 단축
- 작업 실패시, 해당 DAG(태스크)만 재실행 가능 -> 효율 Good

### #DAG

- DAG : Workflow 태스크 → 방향성 비순환 그래프로 정의한 표현 (파이프라인 기본 구조)
- DAG 파일 구성
  (1) 파이썬 코드
  (2) 태스크 집합 + 태스크 간 의존성 + 실행방법(실행주기) + 메타데이터 등등.... 기술

  (Airflow는 DAG구조 식별하기 위래 코드를 파싱함)

### #Airflow

- 구성 (컴포넌트)
  (1) Airflow 스케줄러 : DAG분석, 스케줄 지난 경우 Airflow 워커에 DAG의 태스크 예약
  (2) Airflow 워커 : 예약된 태스크 선택/실행
  (3) Airflow 웹서버 : Airflow 스케줄러에서 분석한 DAG 시각화, 실행 결과 확인 도와주는 주요 인터페이스 제공
  (참고: 12~13p, 그림1.8/그림1.9)

- 사용시 주의점

  (1) Airflow는 반복 or 배치 실행 기능에 초점 -> 스크리밍(실시간 데이터 처리) 부적합

  (2) 동적 태스크 구현 할 수는 있지만, 웹 상 DAG의 가장 최근 버전만 정의해 표현

  (실행 중 구조변화 없는 파이프라인에 적합)

## Chapter 2 : Apache DAG의 구조

**#데이터 수집**

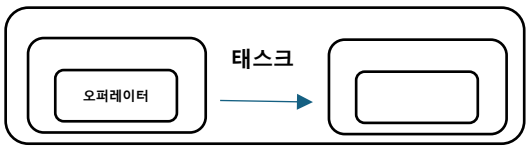
- Launch Library2 사용 (오픈API)

**# DAG(태스크) vs 오퍼레이터 차이점**

- 오퍼레이터 : Airflow에서 단일 작업 수행 역할
  - BashOperator(배시 스크립트 실행시 사용)
  - PythonOperator(파이썬 함수 실행시 사용)
  - EmailOperator(이메일 발송시 사용)
  - ImpleHTTPOperator(HTTP 엔드포인트 호출)
- DAG(태스크) : 오퍼레이터 실행들을 오케스크레이션(조정, 조율) 역할
  - 오퍼레이터 간 의존성 보장
  - 오퍼레이터의 Wrapper or Manager (오퍼레이터 관리 역할)
  - 오퍼레이터 관리 및 사용자에게 상태변경(시작/완료) 표시하는 Airflow 내장 컴포넌트

  (사용자 관점에서, 오퍼레이터 = 태스크 같은 의미로 사용하기도 함)

**#Airflow DAG 작성**




**DAG**

- (장점) 대규모 작업을 개별 태스크로 분할하고, DAG로 형성 가능
- 다중 태스크 병렬처리/다른기술 적용 가능

  : 처음 태스크 -> bash 스크림트 / 두번째 태스크 -> 파이썬 스크립트로 실행 가능
- Airflow는 어떤 태스크가 어떤 DAG에 속하는지 확인 가능
- Airflow코드는 파이썬으로 정의되므로, Workflow와 실행로직을 동일한 스크립트로 관리하는게 편함

  (예제) 태스크 실행순서(download_launches >> get_pictures >> notify)

**[1] DAG 객체 인스턴스 생성**

```
dag = DAG (                                          // 모든 Workflow 시작점
        dag_id = "download_rocket_launches",         // Airflow UI에 표시되는 DAG 이름
        start_date = airflow.utils.dates.days_ago(14), // Workflow 처음 실행되는 날짜/시간
        schedule_interval = None,                    // DAG 자동실행 (X) 설정 (Airflow UI로 실행)
)
```

**[2] 배시 커멘드 실행하기 위해, BashOperator 객체 인스턴스 생성**

```
download_launches = BashOperator (
        task_id = "download_launches",               // 태스크 이름
        bash_command = "curl -o /tmp/launches.json   // 실행할 배시 커멘드
                'https://thespacedevs.com/2.0.0/launch/upcoming'",
        dag=dag,                                     // DAG변수 참조
)
```

- 모든 오퍼레이터에는 task_id 필요
  - ➢ 태스크 실행 시 참조됨
  - ➢ Airflow UI에도 표시

**[3] Pyhon 함수 실행하기 위해, PythonOperator 객체 인스턴스 생성**

```
def _get_pictures() :                                // python 함수
        #디렉터리 존재여부 확인(경로 없으면 디렉터리 생성)   // 함수 내용들
        #URL추출(사진 추출/다운로드/저장/로그)
        #예외처리                                    // (참고: 24~29p)


-- ------------------------------------------------------------------------------------------------

get_pictures = PythonOperator (
        task_id = "get_pictures",                    // 태스크 이름
        python_callable = _get_pictures,             // 실행할 파이썬 함수 지정
```

dag=dag, // DAG변수 참조

)

- PythonOperator 사용 시 필수 사항 2가지
  (1) 오퍼레이터 자신(get_pictures)을 정의해야 함
  (2) Python_callable은 인수에 호출이 가능한 일반함수(_get_pictures)를 가리킴
  (참고 : 편의를 위해 변수 이름을 task_id = get_pictures 동일하게 지음)

**#파이썬 환경에서 Airflow 실행** (참고: 30p)

- pip install apache-airflow
- Airflow 설치 > 메타스토어(모든 Airflow 상태 저장하는 DB) 초기화 > 사용자 생성 > DAG를 DAG 디렉터리에 복사 > 스케줄러/웹 서버 시작
  - 스케줄러/웹 서버 모두 터미널에서 실행되는 프로세스
  - Airflow webserver 백 그라운드에서 실행 or 별도 터미널 창 열어 실행해 두어야 함

**#도커 컨테이너에서 Airflow 실행** (참고: 31p)

- 도커엔진 pc에 설치
  - 그 이후, 명령어 사용해 Airflow 실행 가능
  - 도커에서 Airflow 실행 시, 디렉커리는 호스트 시스템(사용자 pc)이 아닌 컨테이너 내부에 위치

**#Airflow UI** (참고: 32~39p)

- Airflow UI
  - 그래프 뷰 : DAG구조 확인
  - 트리뷰 : 시간 경과에 따른 DAG(태스크) 상태 확인
- 스케줄 간격설정

```
dag = DAG (
        dag_id = "download_rocket_launches",
        start_date = airflow.utils.dates.days_ago(14),
        schedule_interval = "@daily",                    // 하루에 한번 실행 설정
```

- )
- 실패한 태스크 처리
  - 실패한 태스크 클릭 후, 'Clear버튼' 클릭 -> 재실행