Airflow

Computación Tolerante a Fallos

Alumno:

Código:

Profesor(a): Dr. Michel Emanuel López Franco

Sección: D06

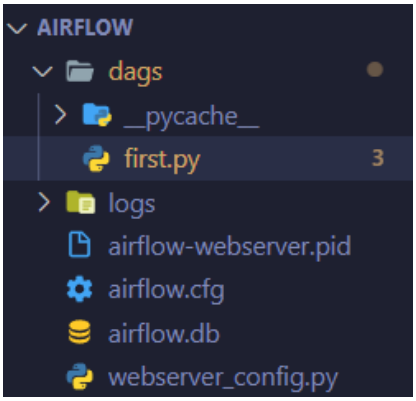
Fecha de Entrega: 03 de octubre de 2022

Apache Airflow

Apache Airflow es un manejador de flujos de trabajo de código abierto desarrollado por Airbnb en Python. Inicialmente, su propósito era gestionar los flujos de trabajo de los empleados de dicha empresa. El objetivo de esta actividad es implementar un programa ETL utilizando Apache Airflow y Python.

El ejemplo que se va a implementar es un programa que realiza peticiones a una API de imágenes, las descarga, y las muestra en pantalla al usuario. Antes de comenzar, es importante mencionar que Airflow no funciona de forma nativa dentro de Windows, por lo que será necesario activar el *Windows Subsystem for Linux* e instalar una terminal Ubuntu desde la Microsoft Store. Habrá que configurar variables de entorno y carpetas para que la instalación funcione correctamente.

Cuando la instalación ha concluido, debemos dirigirnos a la carpeta donde se almacena Airflow y crear otra llamada *dags.* Dentro de esta, se guardaran los archivos que correspondan al flujo de trabajo.



En el archivo *first.py,* importamos las librerías necesarias para trabajar con Airflow y para que el programa pueda realizar las peticiones. La clase DAG es para crear un *Grafo Dirigido Aciclico*, la base de los flujos de trabajo de Airflow, *PythonOperator* se utiliza para definir cada una de las partes que conforma el flujo, y *days_ago* se usa para especificar hace cuanto se tuvo que haber ejecutado.

```python
from datetime import timedelta

from airflow import DAG

from airflow.operators.python import PythonOperator
from airflow.utils.dates import days_ago

import requests, os, random
```

En una lista, guardamos los temas de las imágenes que se podrán mostrar, seguido de la clave de la API. No me molesta mostrarla, ya que después de esta actividad la eliminare.

```python
# Temas de las imagenes.
unsplashQueries = ["history", "life", "wisdom", "art", "office", "nature"]

# Obtener la clave de la API.
apiKey = '5y-QD2AN071LSFQ38VArLTUknHJUN2dmGQiNDDpipiw'
```

Creamos un diccionario denominado *default_args*, que guarda los datos requeridos por el flujo para determinar su modo de trabajo. Puede enviar correos cuando ocurra algún error, establecer el numero de reintentos, y el tiempo que debe pasar entre cada reintento.

```python
default_args = {
    'owner': 'Raul',
    'depends_on_past': False,
    'email': ['raul.ochoa9077@alumnos.udg.mx'],
    'email_on_failure': True,
    'email_on_retry': False,
    'retries': 5,
    'retry_delay': timedelta(minutes=1)
}
```

La primera función, denominada *obtenerDatos*, se encarga de realizar la consulta a la API. Usando *random*, obtenemos un tema aleatorio y formateamos el URL para agregar el tema y la clave. Los datos deben ser convertidos a JSON y finalmente son retornados.

```python
# Extraer
def obtenerDatos ():
    # Solicitamos los datos de la imagen a la API.
    tema = random.choice (unsplashQueries)
    url = f"https://api.unsplash.com/photos/random?query={tema}&client_id={apiKey}"

    peticion = requests.get (url)

    datos = peticion.json()

    return datos
```

Después tenemos la función que transforma el objeto JSON obtenido de la consulta en solo una URL, de forma que la imagen puede ser descargada. Para hacerlo, simplemente nos basamos en la estructura de respuesta incluida en la documentación y accedemos a la propiedad “urls” en su campo “regular”. Para pasarle los datos de la función anterior a esta, usamos las *cross-communications* (xcom); se debe pasar como parámetro la tarea *ti* e utilizar su función *xcoms_pull* para acceder a la función de la que se quieren extraer los datos mediante su ID.

```python
# Transformar
def transformarDatos (ti=None):
    datos = ti.xcom_pull(task_ids="extract")
    # Obtenemos el url de la imagen.
    url = datos["urls"]["regular"]

    return url
```

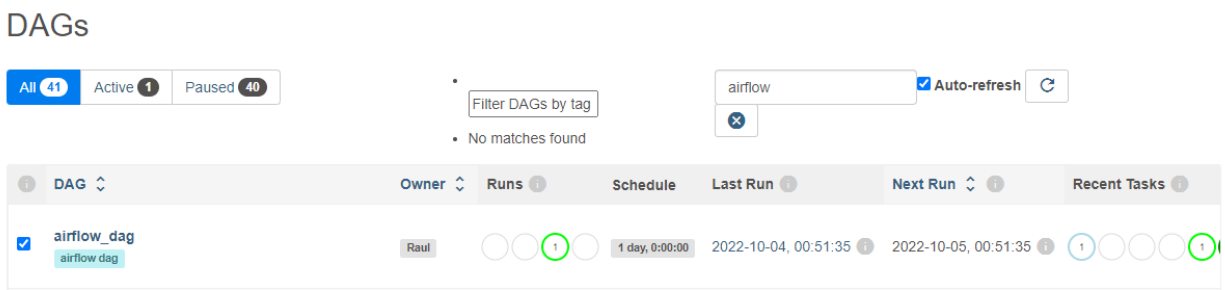
Finalmente, tenemos la parte de cargar los datos a algún lugar. Dentro de la función *almacenarDatos* realizamos una nueva petición, pero esta vez, usando el URL de la imagen. Activamos la bandera *stream* para recibir los datos como sucesiones de bits que pueden ser escritos en un archivo, y de esa forma completamos la descarga. Verificamos si el código de estatus de la petición es distinto a 200 para lanzar una excepción o continuar con el programa. Finalmente, escribimos la sucesiones de bits en el archivo *imagen.jpg* y usamos *system* para ejecutar el archivo y que se muestre en pantalla.

```python
# Almacenar
def almacenarDatos (ti=None):
    imagen = datos = ti.xcom_pull(task_ids="transform")
    # Para evitar tener muchas imagenes descargadas, sobreescribimos el mismo archivo.
    nombreArchivo = "imagen.jpg"
    peticionImagen = requests.get (imagen, stream=True)

    if (peticionImagen.status_code =/= 200):
        raise Exception ("Ocurrio un error al solicitar la imagen a la API.")

    with open (nombreArchivo, "wb") as imagenDescarga:
        for chunk in peticionImagen:
            imagenDescarga.write (chunk)

    os.system ("start imagen.jpg")
```

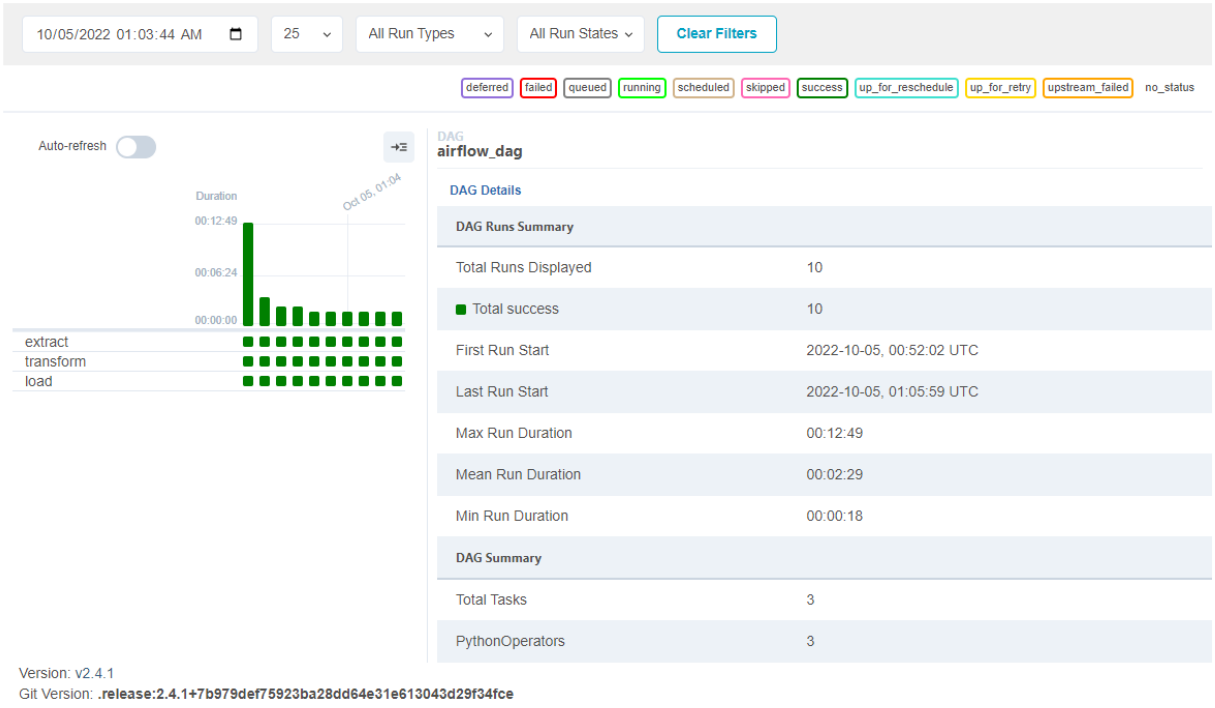
Lo único que resta es crear el DAG; usamos la estructura *with* para hacerlo. Debemos pasar algunos datos al constructor de la clase, como el identificador, el diccionario de argumentos que creamos, una descripción, un intervalo, una fecha de inicio, y los tags. Dentro del *with,* usamos PythonOperator para definir cada una de las funciones como un operador del DAG, y le damos su respectivo ID. Finalmente, usamos los operadores de desplazamiento de bits para indicar el orden del flujo.

```python
with DAG (
    'airflow_dag',
    default_args=default_args,
    description='ETL con Apache Airflow',
    schedule_interval=timedelta(days=1),
    start_date=days_ago(2),
    catchup=False,
    tags=['airflow dag']
) as dag:
    extract_task = PythonOperator(task_id="extract", python_callable=obtenerDatos)
    transform_task = PythonOperator(task_id="transform", python_callable=transformarDatos)
    load_task = PythonOperator(task_id="load", python_callable=almacenarDatos)

    extract_task >> transform_task >> load_task
```

Dentro de la interfaz web de Airflow, el DAG se ve de la siguiente forma:

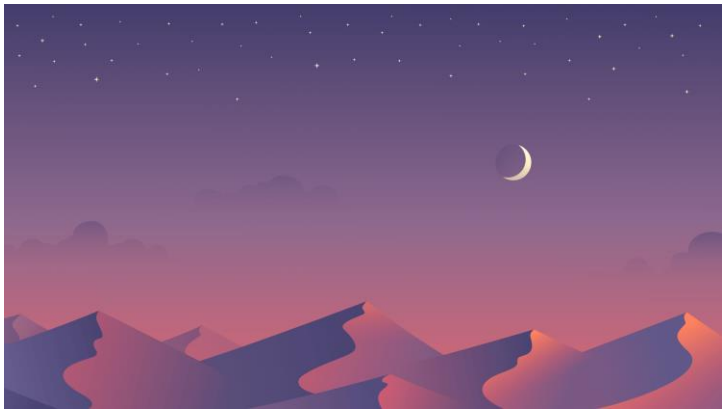


Si ingresamos al apartado, se nos muestra una grafica con las operaciones que ha estado realizando. El verde oscuro indica las que fueron exitosas. Como vemos, se ejecuto 10 veces, y en todas las tres tareas fueron exitosas.



El grafo de las actividades:

Y, por último, la imagen que se mostró:



Conclusiones

Esta actividad fue muy desafiante para mí. No por la complejidad de la aplicación, sino por la instalación de Airflow en mi máquina. Al principio, tenia muchos errores de compatibilidad al usar la terminal de Ubuntu, y los tutoriales no ayudaban. Finalmente, logre combinar ideas de varios videos y foros de internet para hacerlo funcionar e implementar mi programa. Sin duda, aprendí bastante a través del proceso.

En cuanto a la aplicación, fue desarrollada sin problemas, al igual que este reporte de investigación.