Universidad de Guadalajara

Centro Universitario de Ciencias Exactas e Ingenierías

Apache Airflow

Alumno: Bryan De Anda Reyes

Código de alumno: 216195537

Materia: Computación Tolerante a Fallas (D06)

Horario: lunes y miércoles (11:00 – 13:00)

Carrera: INCO

Profesor: Michel Emanuel López Franco

Instrucciones

El objetivo de esta actividad es realizar un ejemplo utilizando el manejo de Apache Airflow con Python. permite la creación, la planificación y el seguimiento de flujos de trabajo a través de la programación informática. Es una solución totalmente de código abierto, muy útil para la arquitectura y la orquestación de circuitos complejos de datos y para el lanzamiento de tareas.

Desarrollo

Para este ejemplo, se analizara el flujo de trabajo de tareas, que en este caso seran tres, estas tareas seran modelos de entrenamiento (a, b, c), al no ser modelos reales su precisión sera generada aleatoriamente, despues, entre estos modelos se seleccionara el que tenga la mejor presición de estas, despues de elegir el modelo, nos indicara si este es presico o impresiso.

Para poner realizar esta practica fue necesario instalar Docker para permitir también la utilización de airflow. Airflow y Docker al solo funcionar con Linux, también fue necesario instalar WSL (Windows Subsystem for Linux).

Después de instalar lo anterior mencionado, para la creación del DAG, lo primero que hice fue crear un archivo my_dag.py el cual es el archivo que se va a codificar para analizar su seguimiento de su flujo de trabajo, este archivo debe crearse en la misma carpeta donde se generó el Docker, la creación del archivo se realiza con el airflow ya en ejecución.

Codificación

Para empezar con la codificación, lo primero que hice fue importar la clase DAG de la librería airflow, también es necesario importar la librería datetime, ya que el flujo de trabajo necesita saber el inicio de su ejecución. Ahora, se crea una instancia de la clase DAG, para esto utilizamos “with DAG() as dag:”, dentro de los parámetros es necesario poner un ID, la fecha de inicio y un intervalo de ejecución, esté ultimo indica con qué frecuencia estará activado el DAG, en este ejemplo se ejecutara cada día.

```python
with DAG("my_dag", start_date=datetime(2022, 1, 1),
    schedule_interval="@daily", catchup=False) as dag:
```

Dentro del with se implementarán cada una de las tareas. Primero, es necesario importar los operadores de airflow para Python, ahora sí, se crea las tareas, dentro de estas se especifican dos argumentos, el ID de la tarea el cual debe ser único para cada una de ellas y el segundo parámetro es una llamada a una función de Python a la que se desea llamar desde esa tarea.

Las siguientes son las primeras tres tareas que son los modelos de entrenamiento A, B, C. Estas tareas tienen su propio ID y llaman a la función “_mejor_modelo()”

```python
modelo_de_entrenamiento_A = PythonOperator(
    task_id="modelo_de_entrenamiento_A",
    python_callable=_mejor_modelo
)

modelo_de_entrenamiento_B = PythonOperator(
    task_id="modelo_de_entrenamiento_B",
    python_callable=_mejor_modelo
)

modelo_de_entrenamiento_C = PythonOperator(
    task_id="modelo_de_entrenamiento_C",
    python_callable=_mejor_modelo
)
```

La función que llaman las tareas anteriores genera una precisión falsa para obtener una simulación del supuesto mejor modelo, esta precisión es un numero aleatorio entre 1 y 10, la tarea que tenga el valor más alto significa que tiene mejor precisión.

```python
def _mejor_modelo():
    return randint(1, 10)
```

Para identificar el modelo con mejor precisión y ejecutarlo, se necesita de BranchPythonOperator, y se crea otra tarea con esta funcionalidad, agregándole su ID y la función a la que llamara.

```
escojer_el_mejor_modelo = BranchPythonOperator(
    task_id="escojer_el_mejor_modelo",
    python_callable=_escojer_el_mejor_modelo
)
```

Para poder crear la función que llama la tarea anterior, es necesario crear otras dos tareas, las cuales indican si en mejor modelo es preciso o impreciso.

```
precisa = BashOperator(
    task_id="precisa",
    bash_command="echo 'precisa'"
)

imprecisa = BashOperator(
    task_id="imprecisa",
    bash_command="echo 'imprecisa'"
)
```

Volviendo a la función que se va a crear para la tarea “escoger_el_mejor_modelo”, en esta primero se obtiene la precisión más alta de las tres tareas y verifica si es mayor a 8, si es así, ese modelo es preciso de lo contrario es inexacto.

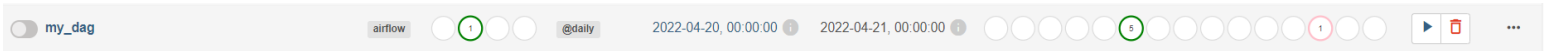
```
def _escojer_el_mejor_modelo(ti):
    accuracies = ti.xcom_pull(task_ids=[
        'modelo_de_entrenamiento_A',
        'modelo_de_entrenamiento_B',
        'modelo_de_entrenamiento_C'
    ])
    best_accuracy = max(accuracies)
    if (best_accuracy > 8):
        return 'precisa'
    return 'imprecisa'
```

Por último, es necesario asignar las dependencias de cada tarea para poder visualizar su flujo correctamente.

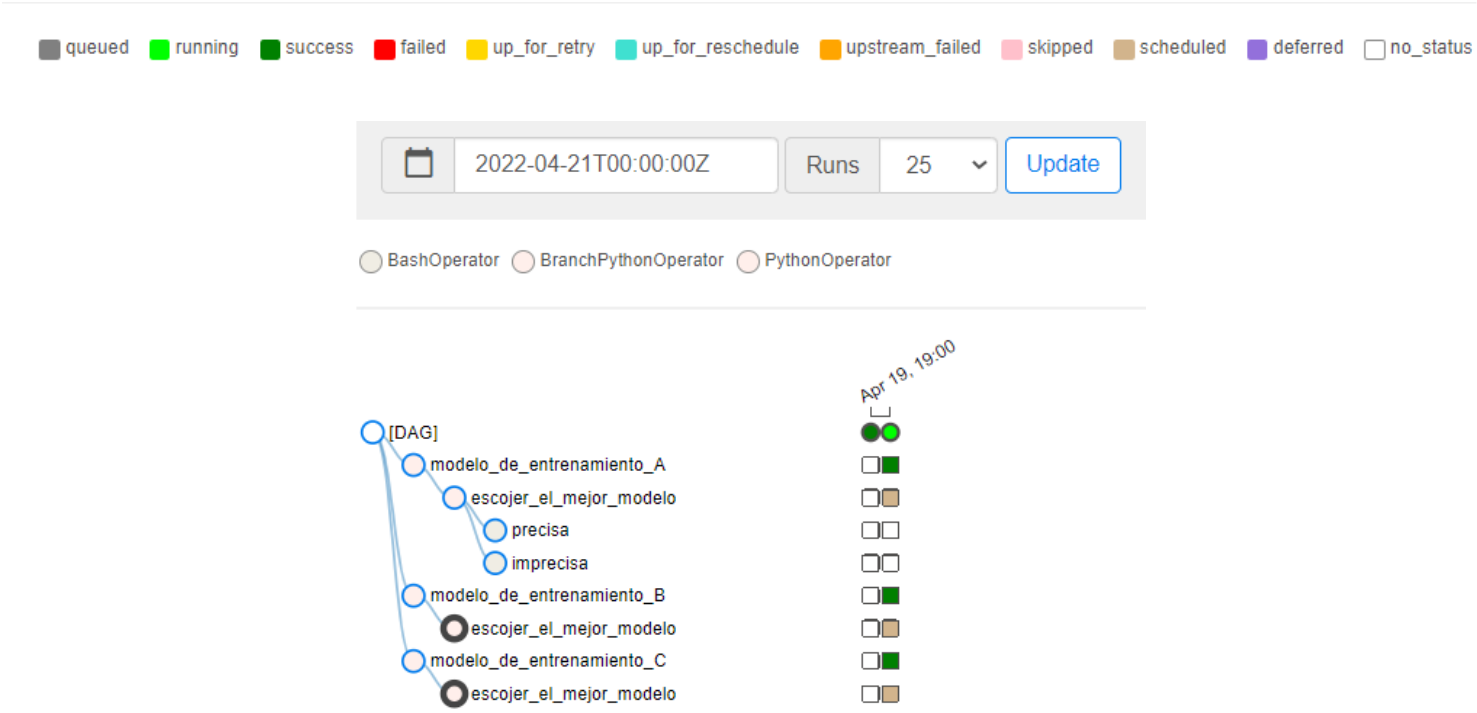
```
[modelo_de_entrenamiento_A, modelo_de_entrenamiento_B, modelo_de_entrenamiento_C] >> escojer_el_mejor_modelo >> [precisa, imprecisa]
```

Ejecución

Para analizar el flujo de trabaja de la simulación se ingresa en el navegador al sitio de Apache Airflow generado por Docker *http://localhost:8080/*, dentro de este sitio nos encontramos con los DAGS que estemos ejecutando y vemos el que creamos llamado “my_dag”.

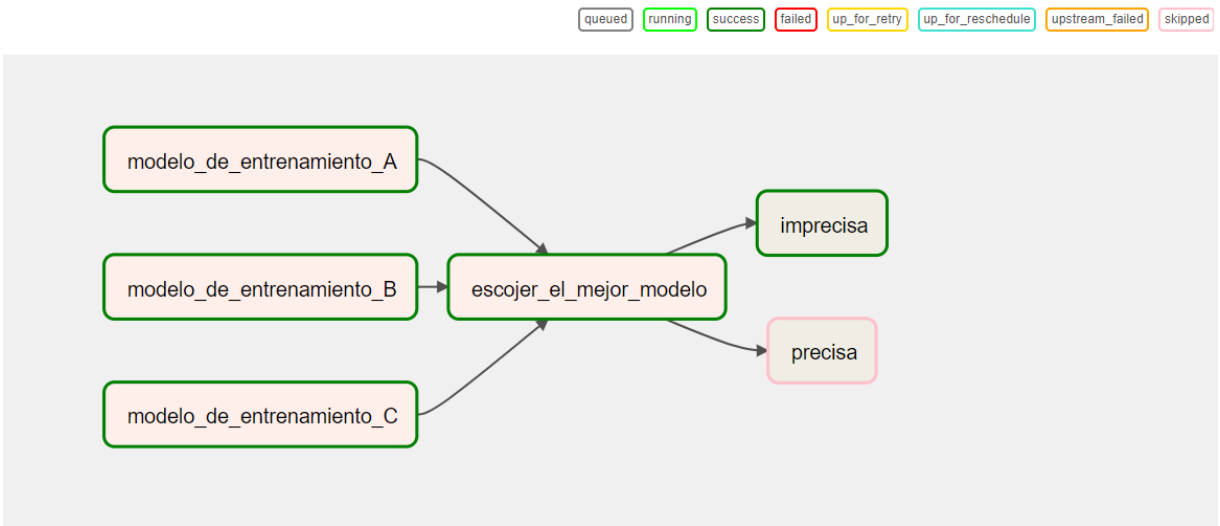


Entramos a este y lo encendemos, podemos observar que inicia con la ejecución, en el árbol, podemos ir observando como va avanzando su ejecución, así nos damos cuenta de que cada uno de los colores representa algo, el verde claro significa nos indica donde se está ejecutando el DAG, el verde oscuro señala que esa tarea la ejecutó con éxito y el color café indica que está en espera.

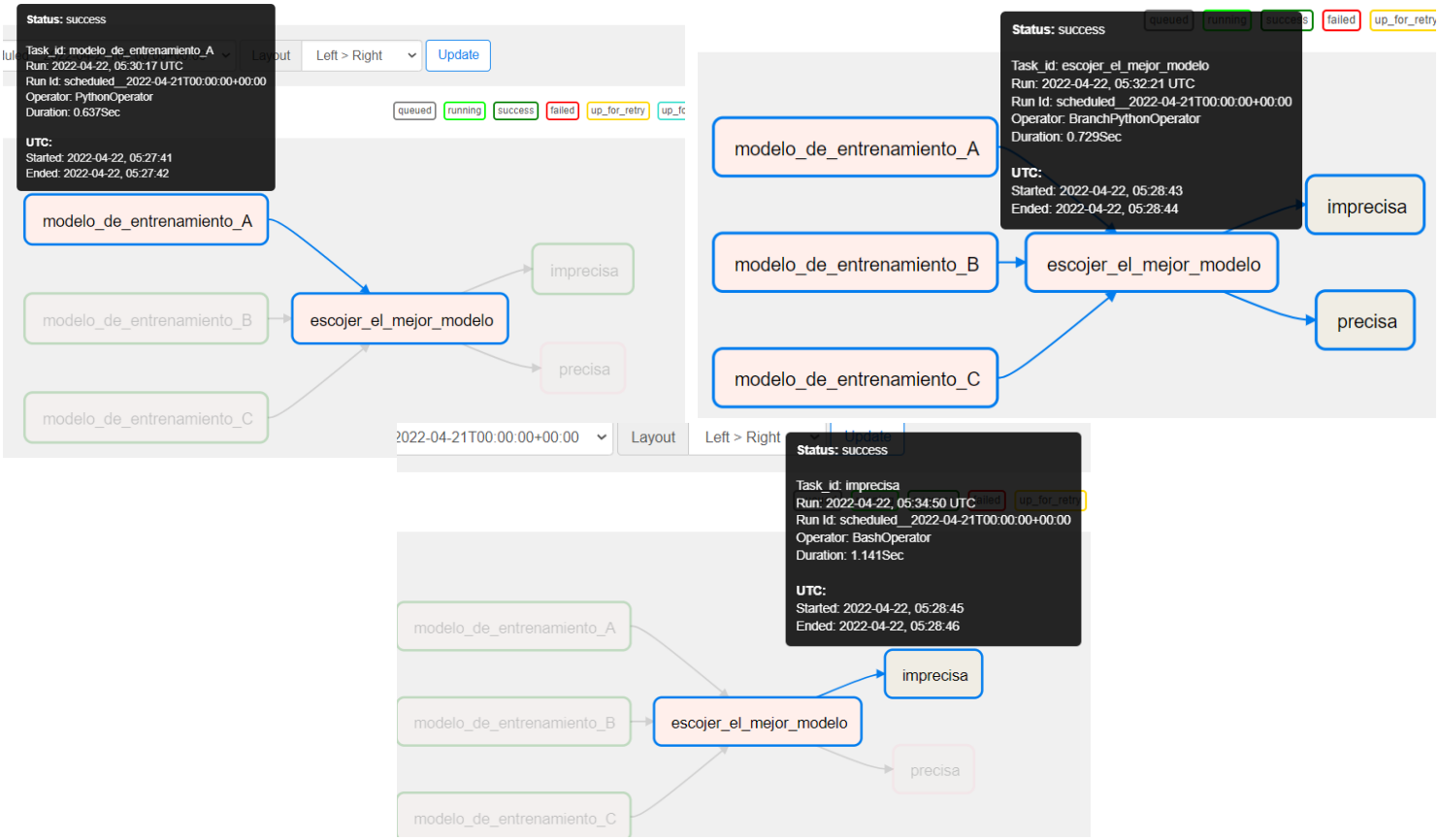


En la siguiente imagen vemos el árbol con la ejecución ya terminada correctamente, aquí podemos apreciar que el modelo de entrenamiento A, fue seleccionado como mejor modelo, pero por otro lado, sigue siendo un modelo impreciso.

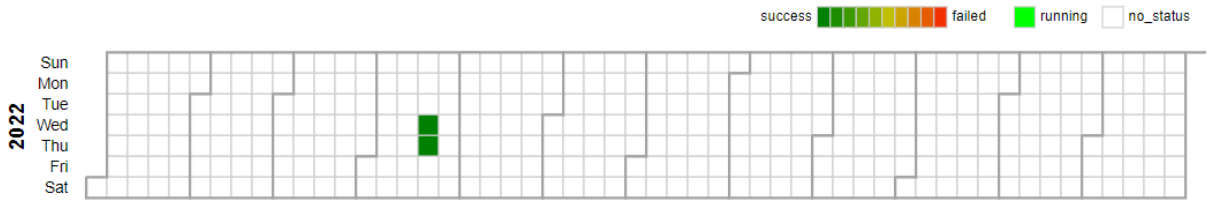
También podemos ver el flujo de trabajo de manera gráfica, mostrando de diferentes colores las tareas que se van ejecutando, de igual manera que en el árbol.



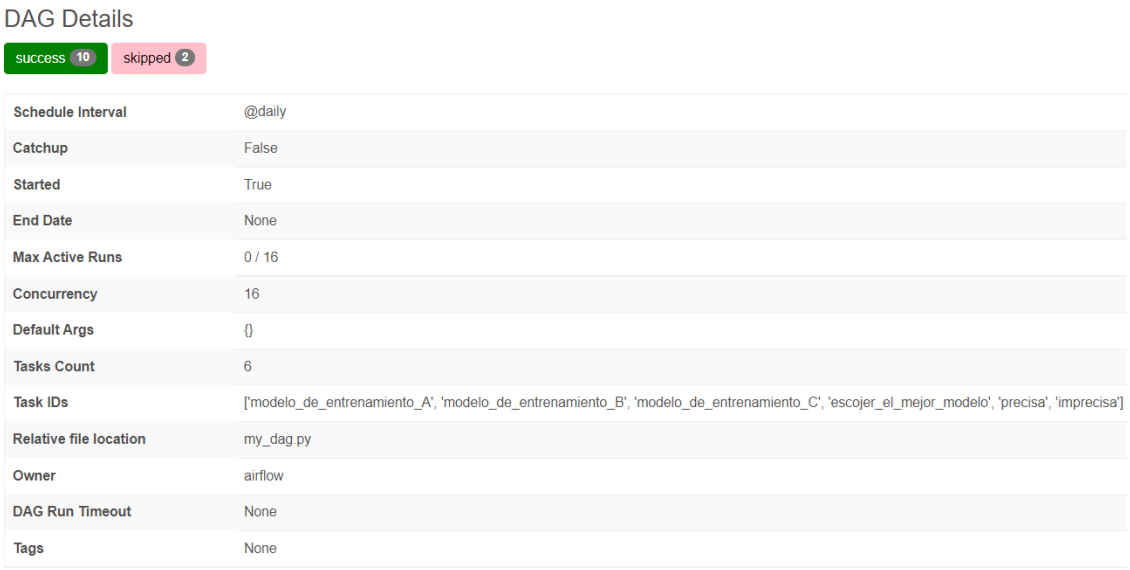
Si seleccionamos cada una de las tareas nos muestra información sobre estas, como su estatus, su identificador, la duración, inicio y fin de ejecución, etc.



Nos muestra el calendario de los días en los que se ejecutó el DAG.



También existen apartados donde podemos ver los diferentes detalles del DAG, el código creado en Python, etc.



Enlace al código en el repositorio: https://github.com/BryanDeAnda/Apache-Airflow.git

Conclusión

Realizar esta actividad me pareció interesante y algo complicada en ciertos aspectos. La instalación de Airflow fue una de las cosas que se me complico ya que al trabajar esta herramienta con Linux fue necesario la instalación de WSL, todo esto siendo algo nuevo para mí.

Por otro lado, hablando sobre el funcionamiento de Apache Airflow, considero que es bastante útil para la creación y seguimiento de diferentes flujos de trabajo. De Airflow podría resaltar varias ventajas, una de ellas es que al ser una plataforma dinámica nos permite ver el fujo de trabajo de diferentes maneras como lo puedes ser en árboles, diagramas o incluso mostrándonos calendarios en los que se ejecuta el flujo, todo esto trabajando con gráficos acíclicos dirigidos, como lo vimos en esta práctica. Otra ventaja que considero nos brinda esta plataforma, es que trabaja con Python.

Bibliografía

C. (2022, 4 marzo). *Apache Airflow : ¿qué es y cómo se usa?* Formación en ciencia de datos | DataScientest.com. Recuperado 2022, de https://datascientest.com/es/todo-sobre-apache-airflow

M. (2021, 2 febrero). *Running Airflow 2.0 with Docker in 5 mins*. YouTube. Recuperado 2022, de https://www.youtube.com/watch?v=aTaytcxy2Ck&t=271s

Python API Reference — Airflow Documentation. (s. f.). Apache Airflow. Recuperado 2022, de https://airflow.apache.org/docs/apache-airflow/stable/python-api-ref.html