1. Arranque del sistema.

Para iniciar correctamente los contenedores, dirígete a la raíz del proyecto y ejecuta el siguiente comando en la terminal:

```
docker-compose up -d
```

Una vez que todos los contenedores estén en funcionamiento, abre tu navegador y accede a **http://localhost:8080**. Esto abrirá la interfaz web de Apache Airflow.

Credenciales de acceso:

- **Usuario:** airflow
- **Contraseña:** airflow

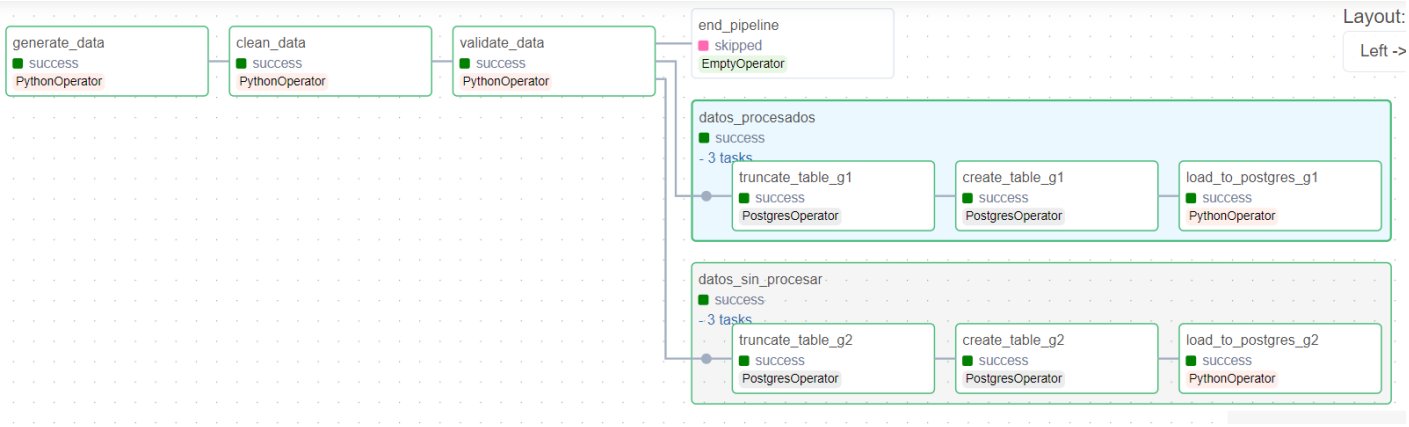
En la pantalla principal, verás la lista de DAGs disponibles. En este caso, solo habrá un DAG, el cual puedes iniciar manualmente desde la interfaz.

Registros de ejecución:
Los logs pueden consultarse directamente en la interfaz de Airflow o accediendo a la carpeta logs dentro del proyecto.

2. Especificaciones técnicas

- **Python**: Lenguaje principal utilizado para el desarrollo del DAG y todas las tareas asociadas.
- **Faker**: Librería de Python utilizada para la generación de datos ficticios de prueba. Permite crear datos realistas y variados con compatibilidad para múltiples formatos (CSV, JSON, SQL, etc.).
- **Pandas**: Librería de Python empleada para la limpieza y manipulación de datos. Facilita la extracción, transformación y carga de datos en diversos formatos de manera sencilla y eficiente.
- **PostgreSQL y SQLAlchemy**: La conexión a la base de datos se realiza a través de Apache Airflow, enlazando con un contenedor de PostgreSQL. Mediante SQLAlchemy, se gestiona el motor de base de datos para insertar los datos limpios en la base de datos.

- **Funcionamiento del pipeline**:



Tareas principales

- **Generate_data**: Generación de datos mediante un operador de Python.
- **Clean_data**: Limpieza de datos utilizando un operador de Python.
- **Validate_data**: Valida los datos tras la tarea **clean_data.** Si los datos no son válidos termina el pipeline inmediatamente en la tarea **end_pipeline**.

Grupo de tareas: Datos procesados

Encargado de gestionar la tabla de usuarios válidos.

- **Truncate_table_g1**: Elimina la tabla **usuarios_validos** si existe.
- **Create_table_g1**: Crea la tabla **usuarios_validos**.
- **Load_to_postgres_g1**: Carga los datos procesados en la tabla **usuarios_validos** de PostgreSQL.

Grupo de tareas: Datos sin procesar

Encargado de gestionar la tabla de usuarios inválidos.

- **Truncate_table_g2**: Elimina la tabla **usuarios_invalidos** si existe.
- **Create_table_g2**: Crea la tabla **usuarios_invalidos**.
- **Load_to_postgres_g2**: Carga los datos sin procesar en la tabla **usuarios_invalidos** de PostgreSQL.

La tabla **usuarios_invalidos** no tiene una **PRIMARY KEY**, ya que los datos provienen del archivo **messy_data.csv**, el cual contiene registros

duplicados. Esta tabla se crea con el propósito de comparar la diferencia entre los datos correctos e incorrectos.

3. Problemas de implementación

Problema 1: Generación de datos

Inicialmente, opté por crear un contenedor de **Ollama**, una herramienta que permite ejecutar modelos de lenguaje (LLM) de forma local. La generación de datos requería realizar una solicitud al servicio de Ollama dentro del contenedor, proporcionando un **prompt** que devolviera una respuesta en el formato deseado.

Sin embargo, surgieron algunas limitaciones:

- Al ejecutarse dentro de un contenedor, el modelo de lenguaje estaba restringido por el tamaño de almacenamiento disponible, lo que impedía el uso de modelos más grandes y potentes.
- Además, la respuesta generada no siempre mantenía el formato correcto y estaba limitada por un número máximo de tokens, lo que restringía la cantidad de filas que se podían generar.

Solución: Opté por utilizar la librería **Faker** de Python, que es una herramienta eficiente para la generación rápida y sencilla de datos ficticios y aleatorios. Esta solución permite crear información estructurada de manera flexible y, junto con **Pandas**, facilita la exportación de los datos a un archivo **CSV** para su posterior uso.

Problema 2: Manejo de fallos en el DAG

Al intentar implementar un bucle en el **DAG** para manejar los fallos y reintentar en caso de error, se generó un error relacionado con la creación de un bucle infinito, lo que podría afectar la estabilidad del flujo de trabajo.

Solución: Para evitar este problema, decidí implementar una **validación de datos** más robusta. Ahora, he configurado un proceso de validación dentro del DAG. Si los datos no son correctos, la validación lanzará una **excepción**, lo que provocará que el pipeline se detenga de inmediato. Esto garantiza que no se carguen datos incorrectos en la base de datos y que el flujo de trabajo termine sin continuar hacia las etapas siguientes en caso de detectar un error en los datos.

Cambio de implementación 1:

Al comienzo, para borrar los datos de las tablas de la base de datos para insertar unos datos nuevos en la nueva ejecución del DAG, borraba la tabla directamente con el comando de SQL:

```
DROP TABLE nombre_tabla;
```

Cada vez que se borra la tabla al comienzo del grupo de tareas, hay que volver a crearla. Para evitar gastar más tiempo y recursos, hice un comando PL/SQL para verificar si la tabla existía, y si existía ejecutaba el comando **"TRUNCATE TABLE nombre_tabla"** borrando así los datos de la tabla. Esta verificación es debido a que este comando da un error si la tabla no existe. El cambio de la implementación es el siguiente:

```
sql=''' DO $$
        BEGIN
            IF EXISTS (SELECT FROM pg_tables WHERE tablename = 'usuarios_validos') THEN
                TRUNCATE TABLE usuarios_validos;
            END IF;
        END $$;
    '''
```

4. Configuraciones especiales

Para probar el uso de **Ollama**, es necesario ejecutar el siguiente comando después de levantar **Docker Compose**:

```
docker exec -it ollama_container ollama pull llama3.1
```

Una vez ejecutado el comando, es necesario **descomentar** el código dentro del archivo **dag.py**, específicamente en la función **ejecutar_generate_data**.

Al ejecutar el DAG nuevamente, se generará un nuevo archivo llamado **prueba.csv** en el directorio **/data**, que contendrá la respuesta del modelo de lenguaje.