Loguear eventos en Airflow

(Data Engineer)

Introducción

Bienvenidos al módulo 5 - Loguear Eventos en Apache Airflow. En el mismo, los objetivos de aprendizaje estarán orientados a incorporar conocimientos generales sobre Apache Airflow y su sistema de logging

Una vez finalizado este módulo estarás capacitado para las siguientes acciones:

- Comprender qué es Apache Airflow y los componentes principales de su arquitectura.
- Comprender el ciclo de vida de una tarea.
- Acceder al archivo de configuración y detectar las variables principales para ajustar el formato del logging.
- Visualizar en la interfaz web los logs que genera un DAG.

Tema 1. Introducción a Apache Airflow

Objetivos

Una vez finalizado este tema estarás capacitado para las siguientes acciones:

- Comprender qué es Apache Airflow y su principio de funcionamiento.
- Comprender los componente principales de su arquitectura
- Comprender el ciclo de vida de una tarea.

¿Qué es Apache Airflow?

Apache Airflow es una plataforma de código abierto para desarrollar, programar y monitorear flujos de trabajo orientados a lotes. El framework Python extensible de Airflow nos permite crear flujos de trabajo que se conectan con prácticamente cualquier tecnología y su interfaz web nos ayuda a administrar el estado de los flujos de trabajo.

Airflow se puede implementar de muchas maneras, desde un solo proceso en su computadora portátil hasta una configuración distribuida para admitir incluso los flujos de trabajo más grandes.

La característica principal de los flujos de trabajo de Airflow es que están definidos mediante código Python, lo cual le brinda las siguientes cualidades:

- **Dinámico** : las canalizaciones o pipelines de Airflow se configuran como código Python, lo que permite la generación de canalizaciones dinámicas.
- **Extensible** : el framework de Airflow contiene operadores para conectarse con numerosas tecnologías. Todos los componentes de Airflow son extensibles para adaptarse fácilmente a nuestro entorno.

- **Flexible**: la parametrización del flujo de trabajo está integrada aprovechando el motor de plantillas de Jinja.

A continuación podemos visualizar una pieza de código que se utiliza en Airflow.

```python
from datetime import datetime

from airflow import DAG
from airflow.decorators import task
from airflow.operators.bash import BashOperator

# A DAG represents a workflow, a collection of tasks
with DAG(dag_id="demo", start_date=datetime(2022, 1, 1), schedule="0 0 * * *") as dag:

    # Tasks are represented as operators
    hello = BashOperator(task_id="hello", bash_command="echo hello")

    @task()
    def airflow():
        print("airflow")

    # Set dependencies between tasks
    hello >> airflow()
```

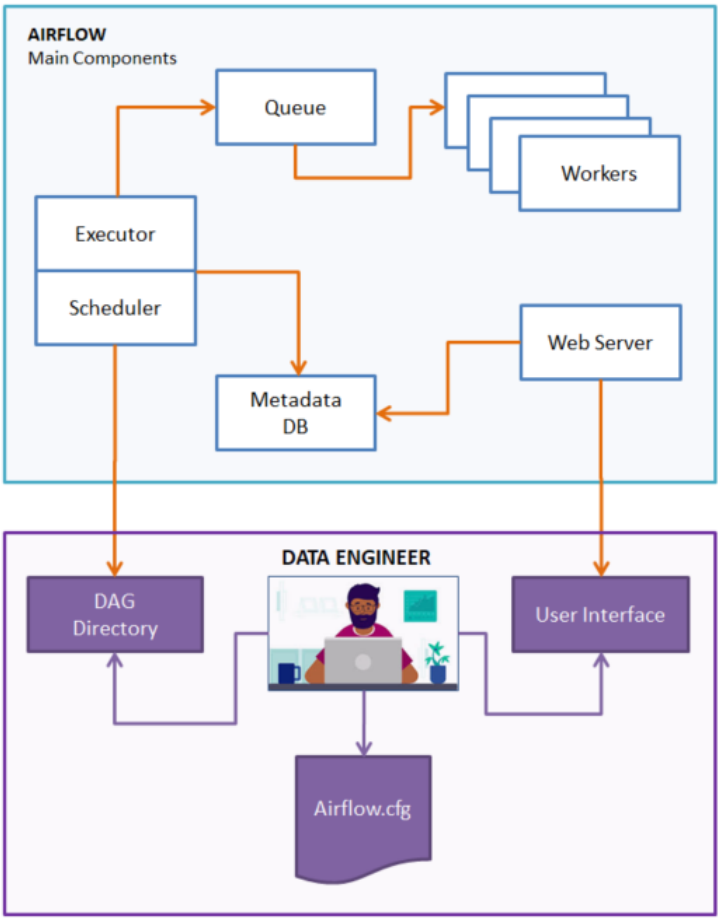
Aquí tenemos:

- **Un DAG** llamado "demo", que comienza el 1 de enero de 2022 y se ejecuta una vez al día. Un DAG es la representación de Airflow de un flujo de trabajo.
- **Dos tareas**, un BashOperator que ejecuta un script Bash y una función de Python definida con el @taskdecorador.
- **Una dependencia entre tareas**, que se indica con el símbolo >> y que controla en qué orden se ejecutarán las mismas.

En resumen, Airflow permite crear, programar y ejecutar flujos de trabajo. Cada flujo de trabajo se representa como un gráfico acíclico dirigido (DAG) que consta de tareas. Estas tareas realizan ciertas operaciones y también pueden tener algunas dependencias entre ellas.

Arquitectura de Airflow

Airflow consta de algunos componentes diferentes que realizan ciertas operaciones y trabajan juntos para permitir que los usuarios diseñen, creen, programen y ejecuten flujos de trabajo. A continuación, repasaremos la arquitectura de Airflow y analizaremos algunos de sus componentes más importantes.



Scheduler (programador)

Esencialmente realiza dos tareas determinadas; Programa y desencadena flujos de trabajo. Además, envía al **executor** (ejecutor) todos los flujos de trabajo programados.

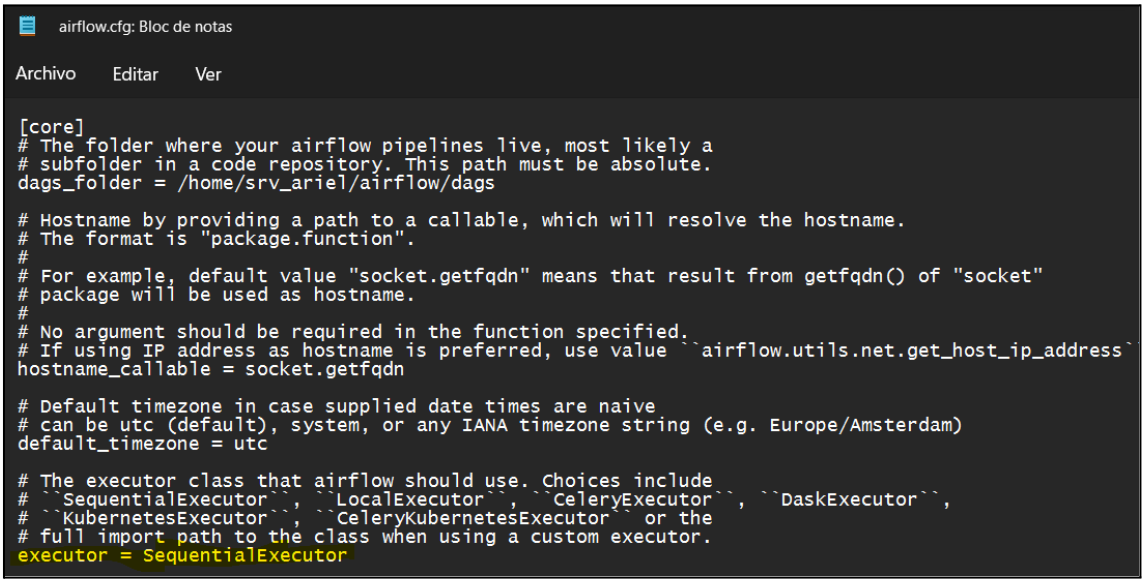
Para determinar si se puede activar alguna tarea, el programador debe ejecutar un subproceso que sea responsable de monitorear la carpeta DAG (que es la carpeta en la que se supone que deben permanecer todos los archivos de Python que contienen los DAG). De

forma predeterminada, el programador realizará esta búsqueda una vez por minuto (pero esto se puede ajustar en el archivo de configuración de Airflow).

El programador usa el ejecutor para ejecutar tareas que están listas.

Executor (ejecutor)

El ejecutor es responsable de ejecutar las tareas. Una instalación de Airflow sólo puede tener un ejecutor en un momento dado. El ejecutor se define en la sección [core] del archivo de configuración de Airflow (airflow.cfg). Por ejemplo:



Existen dos tipos de ejecutores:

- **Locales**: ejecutan tareas localmente, dentro del proceso del programador. Algunos de ellos son:
    - **DebugExecutor**: es una herramienta de depuración y se puede utilizar desde el entorno de de desarrollo integrado (IDE).
    - **LocalExecutor**: ejecuta tareas generando procesos de forma controlada en diferentes modos.
    - **SequentialExecutor:** es el ejecutor predeterminado cuando instala por primera vez airflow. Es el único ejecutor con el que se puede usar sqlite, ya que sqlite no admite conexiones múltiples.

- **Remotos**: ejecutan sus tareas de forma remota (por ejemplo, en un pod dentro de un clúster de Kubernetes), generalmente con el uso de un grupo de trabajadores. Algunos de estos ejecutores son:
    - **CeleryExecutor:** es una de las formas en que puede escalar horizontalmente el número de trabajadores.
    - **KubernetesExecutor:** ejecuta cada instancia de tarea en su propio pod en un clúster de Kubernetes.

Queue (cola)

Una vez que el programador identifica las tareas que se pueden activar, las colocará en una Cola de tareas en el orden correcto en el que se supone que deben ejecutarse.

Los trabajadores (workers) de Airflow extraerán las tareas de la cola para ejecutarlas.

Workers (trabajadores)

Son aquellos que ejecutan las tareas asignadas.

Metadata Database (Base de datos de metadatos)

Es la base de datos utilizada por el ejecutor, el programador y el servidor web para almacenar su estado. De manera predeterminada, una base de datos SQLite se activará, pero Airflow puede usar como su base de datos de metadatos cualquier base de datos compatible con SQLAlchemy. En general, los usuarios tienden a tener una fuerte preferencia por PostgreSQL.

Web Server (Servidor web)

Es un servidor web Flask que expone una interfaz de usuario que permite a los usuarios administrar, depurar e inspeccionar un flujo de trabajo y sus tareas.

Archivo de configuración (airflow.cfg)

Este archivo se encuentra en la ruta /home/[usuario]/airflow/logs. En la documentación oficial de Airflow existe una entrada dedicada exclusivamente a este archivo, donde se pueden visualizar todos los parámetros que se pueden configurar.

Directorio de DAG (DAG Directory)

Es donde se almacenan los gráficos acíclicos dirigidos (DAGS) que consta de tareas y secuencia de ejecución. Esta carpeta se encuentra en la ruta /home/[usuario]/airflow/dags y puede ser editada en el archivo airflow.cfg

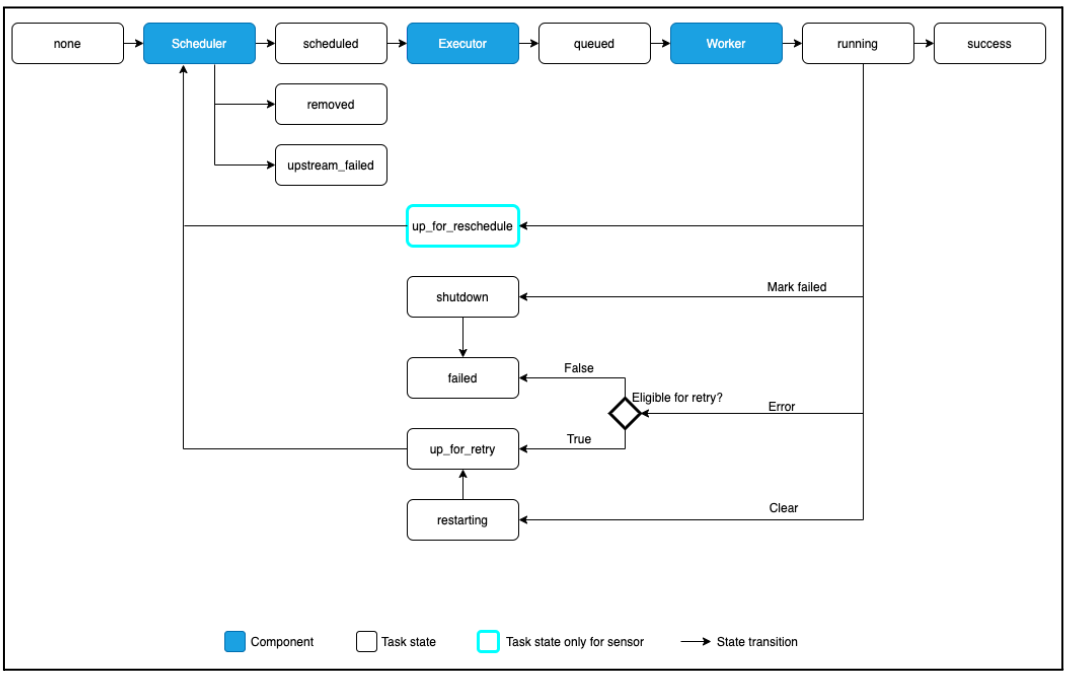
Comando airflow info

El comando airflow info nos proporciona un resumen general de Airflow, donde podemos observar las rutas configuradas para la carpeta DAGS, LOGS, información del sistema operativo, entre otros.

```
srv_ariel@DESKTOP-V8TU0NH:~$ airflow info

Apache Airflow
version                | 2.2.5
executor               | SequentialExecutor
task_logging_handler   | airflow.utils.log.file_task_handler.FileTaskHandler
sql_alchemy_conn       | sqlite:////home/srv_ariel/airflow/airflow.db
dags_folder            | /home/srv_ariel/airflow/dags
plugins_folder         | /home/srv_ariel/airflow/plugins
base_log_folder        | /home/srv_ariel/airflow/logs
remote_base_log_folder |
```

Ciclo de vida de una tarea

En el siguiente diagrama podemos visualizar como Airflow podría asignar diferentes estados a una tarea, durante su ciclo de vida.



- **none**: ninguna tarea se ha puesto en la cola para su ejecución, ya que sus dependencias aún no se cumplen.
- **scheduled**: el programador concluyó que se cumplieron las dependencias de la tarea y que debería ejecutarse.
- **removed**: cuando se detecta un task_id en la base de datos, que no se encuentra asociado a un objeto DAG.
- **upstream_failed**: un trabajo ascendente falló, a pesar de que la regla de activación indicó que era necesario.
- **queued**: se ha asignado un ejecutor a la tarea y está esperando un trabajador.
- **running**: la tarea se está ejecutando en un trabajador (o en un ejecutor local/síncrono).
- **success**: la tarea se completó con éxito y sin fallas.
- **up_for_reschedule**: si el criterio del sensor es Falso, entonces el sensor liberará al trabajador a otras tareas.
- **shutdown**: cuando la tarea se estaba ejecutando, se solicitó que se cerrara desde el exterior de Airflow.
- **failed**: la tarea encontró un error durante la ejecución y no pudo completarse.
- **up_for_retry**: la tarea falló, pero todavía hay intentos disponibles, por lo tanto, se reprogramará.
- **restarting**: cuando el trabajo se estaba ejecutando, se realizó una solicitud externa para que se reinicie.

Tema 2. Logging y monitoreo

Objetivos

Una vez finalizado este tema estarás capacitado para las siguientes acciones:

- Comprender cómo funciona el sistema de logging y monitoreo.
- Acceder al archivo de configuración y editar el formato del logging
- Visualizar logs en el servidor web de Airflow.

Introducción

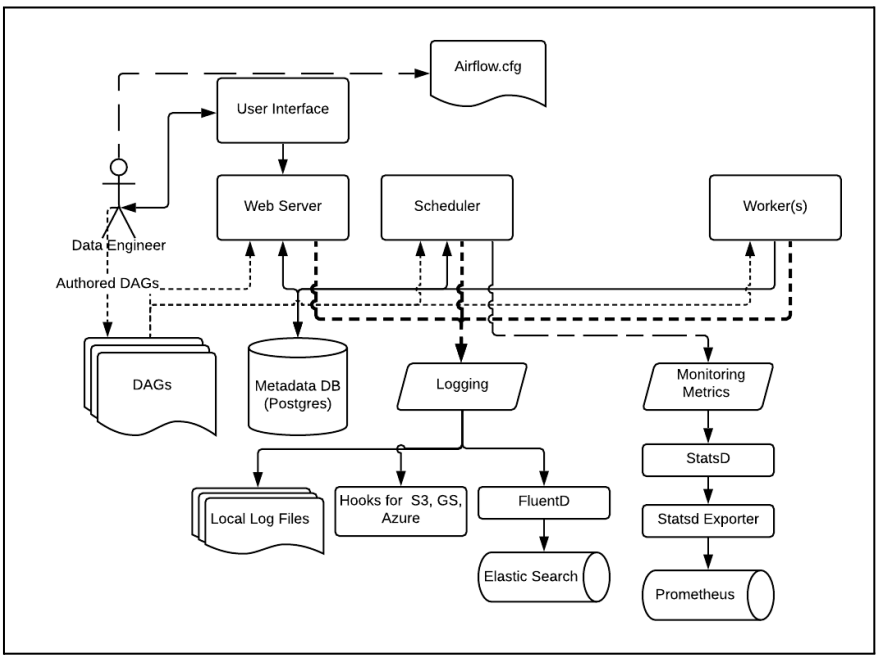
Dado que los pipelines de datos generalmente se ejecutan sin supervisión manual, observar su funcionamiento es fundamental.

Airflow admite múltiples mecanismos de registro o logging, así como un mecanismo integrado para emitir métricas para la recopilación, el procesamiento y la visualización en otros sistemas posteriores. Las capacidades de registro son críticas para el diagnóstico de problemas que pueden ocurrir en el proceso de ejecución de canalizaciones de datos.

Además de las capacidades estándar de registro y métricas, Airflow admite la capacidad de detectar errores en su funcionamiento mediante una verificación de estado. Dado que Airflow generalmente se usa para ejecutar canalizaciones de datos en producción, también admite la notificación de errores en tiempo real a través de la integración con Sentry.

Arquitectura de Logging y Monitoreo

Airflow soporta una variedad de mecanismos de registro y monitoreo, como se muestra a continuación.



De forma predeterminada, el logging se almacena en el sistema de archivos local. Estos incluyen registros del servidor web, el programador y las tareas en ejecución de los trabajadores. Esta configuración es adecuada para entornos de desarrollo y para una depuración rápida.

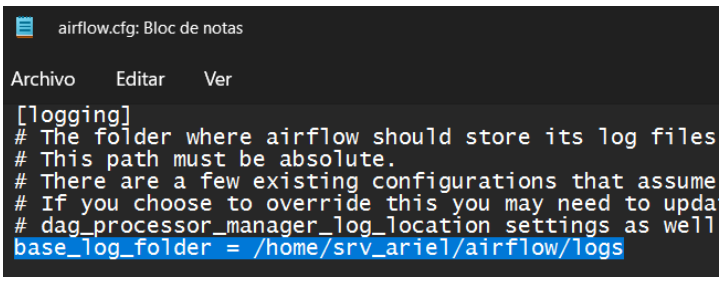
Para las implementaciones en la nube, Airflow también cuenta con controladores aportados por la comunidad para iniciar sesión en el almacenamiento en la nube, como AWS, Google Cloud y Azure.

La configuración y las opciones de registro se pueden especificar en el archivo de configuración de Airflow, que debe estar disponible para todo el proceso de Airflow: servidor web, planificador y trabajadores.

Registro de tareas

Airflow escribe registros para cada tarea de manera que nos permite ver los registros por separado en la interfaz de usuario

Podemos especificar el directorio donde Airflow deposita los logs editando la ruta de la variable **base_log_folder.**



También podemos ajustar las siguientes variables, entre otras:

log_filename_template: formato de nombres y rutas que genera Airflow para cada tarea que se ejecuta.

```
# Formatting for how airflow generates file names/paths for each task run.
log_filename_template = {{ ti.dag_id }}/{{ ti.task_id }}/{{ ts }}/{{ try_number }}.log
```

log_format y simply_log_format: formato de la línea de log

```
# Format of Log line
log_format = [%%(asctime)s] {%%(filename)s:%%(lineno)d} %%(levelname)s - %%(message)s
simple_log_format = %%(asctime)s %%(levelname)s - %%(message)s
```

Utilización de loggers en Airflow

A los registros de cada tarea que escribe Airflow al procesar un DAG, podemos agregar nuestro propio logger.

Airflow define 4 loggers por defecto: **root**, **flask_appbuilder**, **airflow.processor** y **airflow.task.**

Para utilizarlo, lo incorporamos en nuestro DAG

```
#Logging setup
logger = logging.getLogger("airflow.task")
```

y luego lo utilizamos de acuerdo a nuestras necesidades.

```
#Logging message
logger.info("...Archivo procesado correctamente...")
```

En caso de que necesitemos generar nuestro propio logger, ajustando el nivel de severidad, handlers y formatos específicos, podemos realizarlo de la siguiente manera.

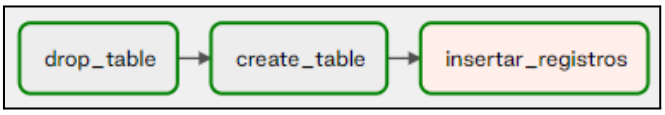
```
#Logging setup
logger = logging.getLogger("custom_logger")
level = logging.getLevelName('INFO')
logger.setLevel(level)

#Format
format = logging.Formatter('%(asctime)s - %(levelname)s -%(levelno)s - %(message)s')

#Handler 1 - Stream Handler (INFO)
stream_handler = logging.StreamHandler()
stream_handler.setLevel(logging.INFO)
stream_handler.setFormatter(format)
logger.addHandler(stream_handler)

#Handler 2 - File Handler (INFO)
file_handler = logging.FileHandler('/usr/local/airflow/include/archivos_tmp/custom_log.log')
file_handler.setLevel(logging.INFO)
file_handler.setFormatter(format)
logger.addHandler(file_handler)
```

Ejemplo: Visualización de un log

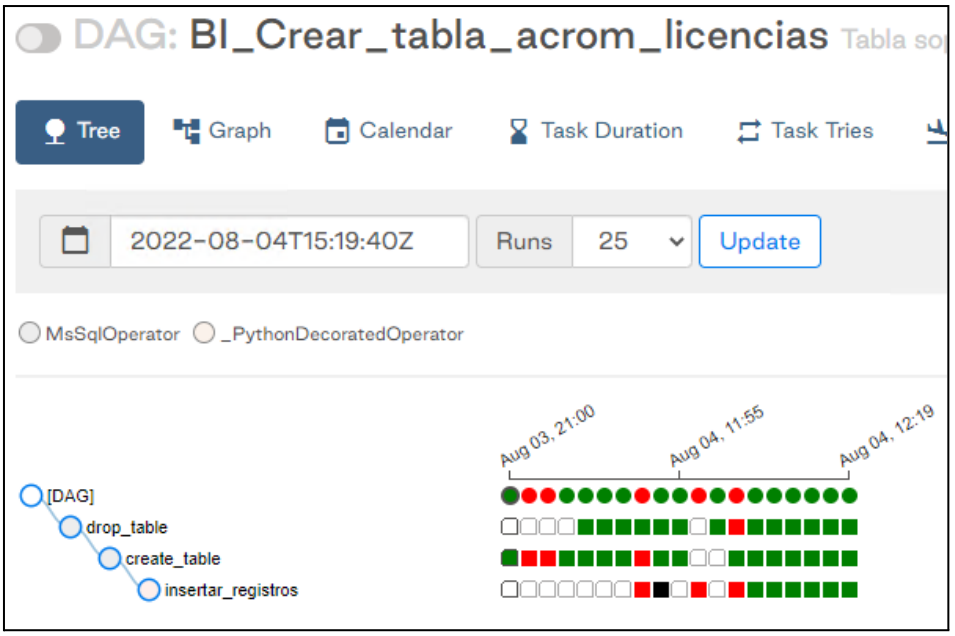
Para este ejemplo hemos creado el siguiente DAG:

BI_Crear_tabla_acrom_licencias

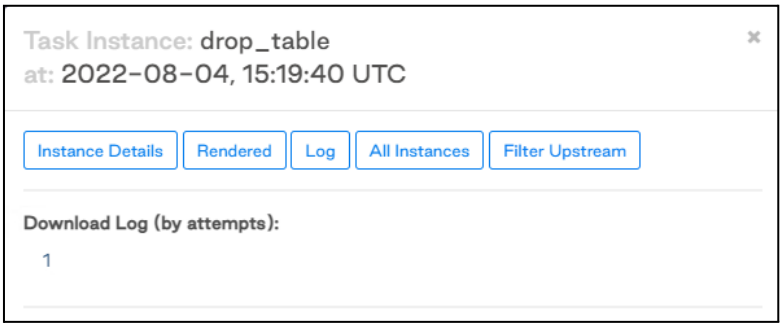
Este DAG contiene tres tareas como se observa en la siguiente imagen.



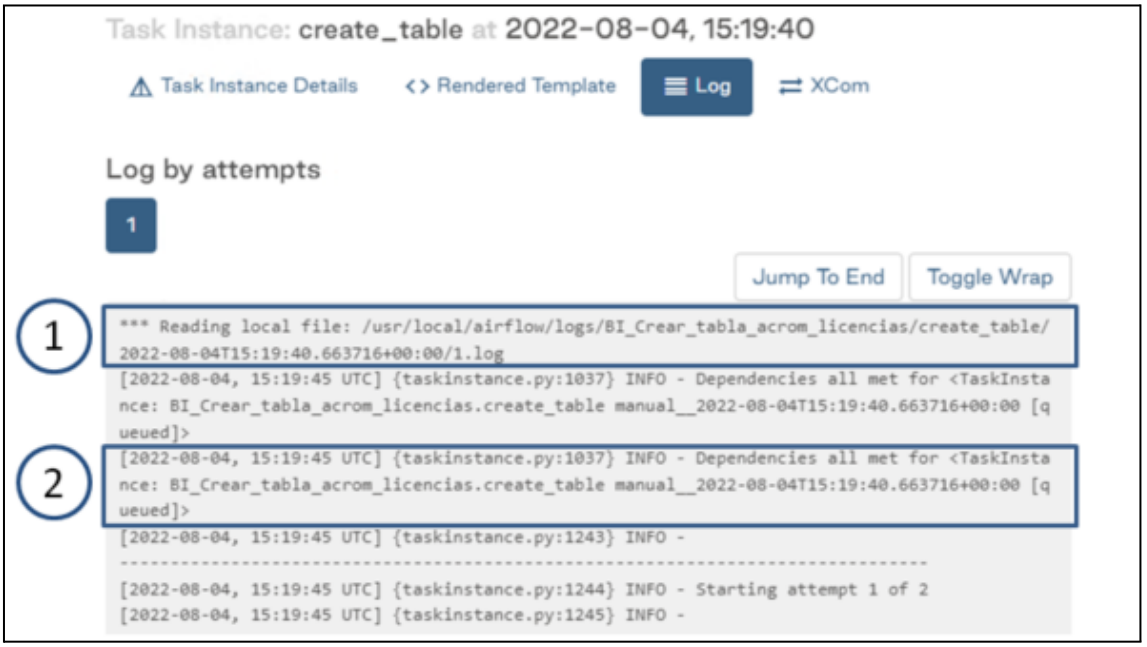
Una vez ejecutado, podemos visualizar los logs generados.



Si hacemos click en el último cuadrado verde de la tarea "drop_table", se abrirá el siguiente menú



Si hacemos click en el botón “**Log**”, accederemos a los logs de esa tarea, que se encuentran detallados en el panel “**Log by attempts**”.



En el recuadro n° 1 podemos ver que se generó el siguiente archivo log:

```
BI_Crear_tabla_acrom_licencias/drop_table/2022-08-04T15:19:40.663716+00:00/1.log
```

Podemos notar que la estructura de ese nombre se corresponde con la definida para la variable **log_filename_template** en el archivo **airflow.cfg**.

Recordando la definición de la variable:

log_filename_template = {{ ti.dag_id }}/{{ ti.task_id }}/{{ ts }}/{{ try_number }}.log

Podemos comprobar:

- ti.dag_id = BI_Crear_tabla_acrom_licencias
- ti.task_id = create_table
- ts: 2022-08-04T15:19:40.663716+00:00
- try_number: 1

En el recuadro n° 2 podemos ver la línea de log que se escribió en el archivo log que vimos en el recuadro n° 1.

```
[2022-08-04, 15:19:42 UTC] {taskinstance.py:1037} INFO - Dependencies all met for <TaskInstance: BI_Crear_tabla_acrom_licencias.drop_table manual__2022-08-04T15:19:40.663716+00:00 [queued]>
```

El formato de esa línea es el que se especifica en la variable **log_format** del archivo **airflow.cfg**.

Recordando la definición de la variable:

log_format = [%%(asctime)s] {%%(filename)s:%%(lineno)d} %%(levelname)s - %%(message)s

Podemos comprobar:

- [%%(asctime)s]: [2022-08-04, 15:19:42 UTC]
- filename: taskinstance.py
- lineno: 1037
- levelname: INFO
- message: Dependencies all met for <TaskInstance: BI_Crear_tabla_acrom_licencias.drop_table manual__2022-08-04T15:19:40.663716+00:00 [queued]

Si nos dirigimos al archivo taskinstance.py podemos visualizar como la función are_dependencies_met, genera el log que luego vemos en el panel de logs.

taskinstance.py 9+

Ubuntu-20.04 > home > srv_ariel > .local > lib > python3.8 > site-packages > airflow > models > taskinstance.py > ...

```
1003            return self.get_previous_start_date(state=State.SUCCESS)
1004
1005        @provide_session
1006        def are_dependencies_met(self, dep_context=None, session=None, verbose=False):
1007            """
1008            Returns whether or not all the conditions are met for this task instance to be run
1009            given the context for the dependencies (e.g. a task instance being force run from
1010            the UI will ignore some dependencies).
1011
1012            :param dep_context: The execution context that determines the dependencies that
1013                should be evaluated.
1014            :type dep_context: DepContext
1015            :param session: database session
1016            :type session: sqlalchemy.orm.session.Session
1017            :param verbose: whether log details on failed dependencies on
1018                info or debug log level
1019            :type verbose: bool
1020            """
1021            dep_context = dep_context or DepContext()
1022            failed = False
1023            verbose_aware_logger = self.log.info if verbose else self.log.debug
1024            for dep_status in self.get_failed_dep_statuses(dep_context=dep_context, session=session):
1025                failed = True
1026
1027                verbose_aware_logger(
1028                    "Dependencies not met for %s, dependency '%s' FAILED: %s",
1029                    self,
1030                    dep_status.dep_name,
1031                    dep_status.reason,
1032                )
1033
1034            if failed:
1035                return False
1036
1037            verbose_aware_logger("Dependencies all met for %s", self)
1038            return True
```

Cierre

Durante esta unidad aprendimos sobre el registro de eventos en Apache Airflow.

Comenzamos describiendo el funcionamiento de Airflow, los principales componentes de su arquitectura y el ciclo de vida de una tarea.

Posteriormente aprendimos sobre el funcionamiento del sistema de logging y monitoreo, donde accedimos al archivo de configuración y pudimos visualizar las principales variables que establecen los formatos de registro.

Finalmente realizamos la exploración de logs en un DAG, donde pudimos comprobar cómo se aplicaron correctamente las configuraciones.

Quedamos a disposición de las consultas que puedan surgir.

Referencias

Documentation | **Apache Airflow.** Recuperado de: https://airflow.apache.org/docs

tiimiit by