Data Pipelines with Apache Airflow

Bas P. Harenslak and Julian Rutger de Ruiter

Resources

Online Book: livebook.manning.com/book/data-pipelines-with-apache-airflow/welcome/v-6

Summary: [PART 1 GETTING STARTED]

1.1 INTRODUCING DATA PIPELINES

- **Apache Airflow** is a batch-oriented framework for building data pipelines by stitching together many different technologies. Think of it as a spider (orchestrator and coordinator) in a web (of different distributed systems). It is not a data processing tool in and of itself.
- A **data pipeline** is several tasks executed in a particular order to achieve a result.
  - Dependencies are represented by **directed acyclic graphs** (DAGS), which consist of tasks as nodes connected by directed edges. Acyclic is necessary to prevent circular dependencies that create ambiguity in execution.
    - The **graph algorithm** states: for each uncompleted task and for each edge pointing to these tasks, check if the upstream task is completed. If so, add the task to the execution list. Repeat until all tasks in the graph have been completed.
    - The graph algorithm has advantages over a linear script as it allows independent branches to be run in parallel rather than sequentially. Also if the pipeline fails, the *entire* script would need to be rerun whereas the graph would only rerun failing tasks.

1.2 INTRODUCING AIRFLOW

- DAGs are defined by Python script describing the structure for the set of tasks. Typically one file corresponds to one DAG and it includes an execution interval for scheduling. This provides flexibility such as having Python code dynamically generate optional tasks or generate entire DAGs using external metadata.
- Airflow has three main components:
  - The **scheduler**: parses DAGs, checks dependencies and checks scheduled intervals before passing execution queue to Airflow workers. Intervals may be fixed (hour, day, week) or take more complicated cron-like expressions.
  - The **workers**: execute tasks, aka “do the actual work”.
  - The **webserver**: user interface where DAGs are visualized and results are returned.
  - In a nutshell:
  - The user writes DAG files that are stored in the Airflow metastore (database) and read in by the scheduler.
  - The scheduler checks if the specified interval has passed, checks dependencies and if valid, passes tasks to the execution queue.
  - The workers pick up the queue, execute the tasks and write results to the metastore.
  - The web server reads results from the metastore and surfaces them in the webserver, which is monitored by the user.
  - The webserver has a **graph view** that visualizes the DAG, and a tree view that shows historical executions. One column in the **tree view** shows a single execution of the DAG, where each row is the status of executions for a single task.
    - Failed tasks are retried and eventually raised as failures. Users inspect the logs through the tree view in the webserver.

- The scheduler also provides details about the last (and expected next) DAG execution so that time may be split into discrete intervals on which the DAG is run.
  - This feature allows for **incremental pipelines** that only process "new/delta" data rather than the entire dataset, which has time and cost benefits. It also enables **backfills or rebuilds** (new DAGs on historical scheduled intervals) for reprocessing entire datasets.
    - Azkaban, for example, does not allow for backfills.

1.3 WHEN TO USE AIRFLOW

- Reasons to choose Airflow: allows for complex pipelines, easily extendable to integrate many different systems, can backfill, and easy monitoring of results through web interface. It is also open source.
- Reasons not to choose Airflow: intended for batch tasks and not streaming, requires Python experience, DAGs require engineering rigour to be maintainable as they become complex. Also, although Airflow can handle dynamic DAGs, web interface only shows most recent definition and thus does not favour pipelines that change tasks with every run.

2.0 ANATOMY OF AN AIRFLOW DAG

- The example in this chapter is a space launch enthusiast that wants to download photos of upcoming rocket launches, which are available at a free API: https://ll.thespacedevs.com/2.0.0/launch/upcoming
- The pipeline is split into three tasks: (1) grab the raw API data using a bash operator and save to a file (2) read in the raw data file and use the image link field to download the photos, and (3) notify the user. Note the number of tasks is an engineering design choice; there could be more or less based on your preference.
- The DAG file has the following components:

  A. Instantiating the DAG object

  - The dag_id is arbitrary naming and no schedule interval means that the DAG needs to be kicked off manually from the UI. To run daily, change this to `"@daily"`.

```
dag = DAG(
  dag_id="download_rocket_launches",
  start_date=airflow.utils.dates.days_ago(14),
  schedule_interval=None,
)
```

  B. Defining the API fetch task

```
download_launches = BashOperator(
   task_id="download_launches",
   bash_command="curl -o /tmp/launches.json 'https://launchlibrary.net/1.4/
                launch?next=5&mode=verbose'",
   dag=dag,
)
```

  C. Writing the photo download operator & defining the photo download task

  - The operator is defined as a function `def _get_pictures()` which reads in the result of the `download_launches` task and downloads the photos using the `image_url` field. Full operator not shown here for brevity.

- Although tasks and operators are often used interchangeably, there is a difference. An operator is responsible for a single piece of work and the task is a wrapper to manage the execution. The task below manages the `_get_pictures` operator.

```
get_pictures = PythonOperator(
   task_id="get_pictures",
   python_callable=_get_pictures,
   dag=dag,
)
```

D. Defining the notify task

```
notify = BashOperator(
   task_id="notify",
   bash_command='echo "There are now $(ls /tmp/images/ | wc -l) images."',
   dag=dag,
)
```

E. Setting the order of task execution

- rshift (">>") indicates a dependency, where `_get_pictures` will not be executed unless `download_launches` is completed successfully.

```
download_launches >> get_pictures >> notify
```

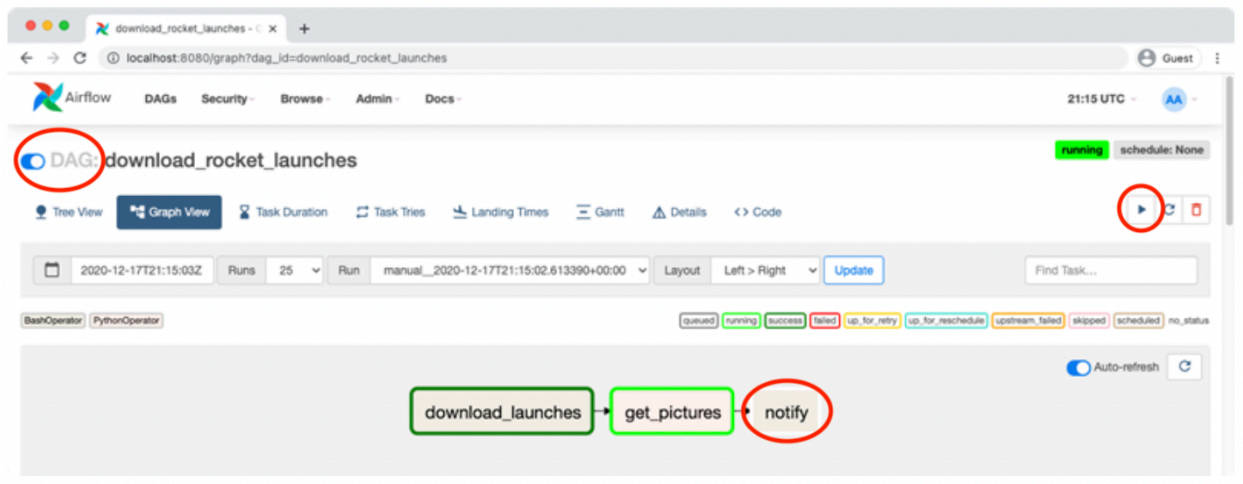
2.3 RUNNING A DAG IN AIRFLOW

- To run Airflow in a **Python environment**:
  - Install airflow as a Python package: `pip install apache-airflow`
  - Save the DAG file from above as `rocket_launches.py`
  - Initialize the metastore: `airflow db init`
  - Create a user: `airflow users create —-username admin —-password admin —-firstname Anonymous —-lastname Admin —-role Admin —-email admin@example.org`
  - Copy the DAG file into DAG directory: `cp download rocket_launches.py ~/airflow/dags/`
  - Start the scheduler and webserver: `airflow webserver & airflow scheduler`
    - Note that the scheduler and webserver are both continuous processes so you may want to open a second terminal.
  - Open http://localhost:8080, log in with admin as the password and username to view Airflow.
- To run Airflow in **Docker containers**:

```
docker run \
-ti \
-p 8080:8080 \
-v /path/to/dag/rocket_launches.py:/opt/airflow/dags/rocket_launches.py \
--entrypoint=/bin/bash \
--name airflow \
apache/airflow:2.0.0-python3.8 \
-c '( \
airflow db init && \
airflow users create --username admin --password admin --firstname Anonymous
--lastname Admin --role Admin --email admin@example.org \
); \
airflow webserver & \
airflow scheduler \
'
```

- Docker containers create isolated environments on the operating system level. This allows containers to contain only a set of Python packages and avoid dependency clashes. Running Docker containers requires an installed Docker Engine.
- Open http://localhost:8080, log in with admin as the password and username to view Airflow.

- Once the webserver is up and running, make sure the DAG is toggled *On* on the top-left.
- Since the DAG is unscheduled, kick off a manual run with the play button on the top-right.
  - The DAG progress is colour-coded. Due to the dependency, notice that `get_pictures` will only start running when `download_launches` is complete.
- When the DAG completes, click individual tasks and then select *Logs* to inspect the logs. In the `rocket_launches.py` file, the `notify` stage had an output log that may be viewed here.
- Finally to view the actual images, navigate to `/temp/images`
  - Note that if you used a Docker container, the directory exists in the container and not your host system, so you need to get to the container first by running: `docker exec -it airflow /bin/bash`



- If a task fails, the cell will appear as red in the tree view on the webserver. The logs may be inspected for errors. To retry the task, click on the cell and select *Clear* to rerun that task. It is unnecessary to restart the entire workflow; Airflow can restart from the failure point and onwards.

3.0 SCHEDULING IN AIRFLOW
4.0 TEMPLATING TASKS USING THE AIRFLOW CONTEXT
5.0 DEFINING DEPENDENCIES BETWEEN TASKS

Summary: [PART 2 BEYOND THE BASICS]
Summary: [PART 3 AIRFLOW IN PRACTICE]
Summary: [PART 4 IN THE CLOUDS]