Data Pipelines with Apache Airflow

Bas P. Harenslak
Julian R. de Ruiter

MANNING

MEAP Edition
Manning Early Access Program

Data Pipelines
with Apache Airflow
Version 1

welcome

Thank you for purchasing the MEAP for *Data Pipelines with Apache Airflow*. We hope the book in its current state is already valuable to you and aim to make the final version of the book even better using your feedback!

This book aims to provide an in-depth introduction for data-savvy professionals to Airflow, a tool/framework that helps you develop complex data pipelines consisting of many tasks and (possibly) spanning different technologies. When we started writing our first Airflow pipelines it was a relief to see a simple Python script gluing together various tasks and handling the complex logic of dependencies, retries, logging, and such.

The Apache Airflow framework holds many possible options for writing, running, and monitoring pipelines. While this gives a lot of freedom to define pipelines in whichever way you like, it also results in no single good or the best way to do so. This book aims to provide a guide to the Airflow framework from start to end, together with best practices and lessons learned from our experience of using Apache Airflow.

As Airflow itself is written in Python, some experience with programming in Python is assumed. Besides this, it helps to have some experience in the data field, as Airflow is merely an orchestration tool for coordinating technologies - it is not a data processing tool in itself. As such, you will need to have some knowledge of the tooling that you are trying to coordinate. The data field is fast and always changing - hence it helps to have experience in the field to quickly understand new technologies and how to fit them in your data pipelines.

Part 1 of the book covers the basics everybody should know of Airflow - the building blocks of the framework. Part 2 provides a deep dive for more advanced users and includes topics as developing and testing custom operators. And part 3 will examine how to run Airflow in production - doing CI/CD, scaling out, security, and more.

We hope you enjoy the book and that it will enjoy a prominent position on your physical bookshelf. We encourage you to provide feedback in the liveBook discussion forum. Any feedback is greatly appreciated and will help improve the book.

— Bas Harenslak & Julian de Ruiter

brief contents

1 Meet Apache Airflow

This chapter covers:

- **What is Apache Airflow**
- **What problems does Airflow solve**
- **Is Airflow the right tool for you**

People and companies are continuously becoming more data-savvy and are developing data pipelines as part of their daily business. Data volumes have increased substantially over the years, rising from mere megabytes per day in early applications to the petabyte-per-day datasets that are encountered today. We can manage the data deluge though.

Some pipelines use real-time data, and others use batch data. Either approach has its own benefits. Apache Airflow is a platform for developing and monitoring batch data pipelines.

Airflow provides a framework to integrate data pipelines of different technologies. Airflow workflows are defined in Python scripts, which provide a set of building blocks to communicate with a wide array of technologies.

Think of Airflow as the spider in a web; it controls systems as a distributed architecture. It is not a data processing tool in itself, but orchestrates processes doing so. This book isolates and teaches parts of that process in the sprawling web. We will examine all components of Airflow in detail. First, before we breakdown the aerial view of the Airflow's architecture and determine if this tool is right for you, let's do a quick overview of workflow managers to align our working mindset.

1.1 Introducing workflow managers

Many computational processes consist of multiple actions that need to be executed in a specific sequence, possibly at a regular interval. These processes can be expressed as a graph of tasks, which defines (a) which individual actions make up the process and (b) in which order these actions need to be executed. These graphs are often called workflows.

Both computational and physical processes can often be thought of a series of tasks that achieve a specific goal when executed the correct order. In this view, tasks are small pieces of work that form small parts of the whole, but together make-up the steps required to produce the desired result. Following this thought, one way of implementing a (software) process is to simply encode these tasks in a linear script that executes tasks one-after-the-other, as shown in the example listing below.

However, this approach makes poor use of our resources if tasks can be executed in parallel, as it is only capable of executing one task at a time. Moreover, if any of the tasks fails, the entire script fails. This means that to re-execute the failing task, we need to restart the process, which involves re-executing tasks that may have executed successfully. Finally, by itself, this approach does not give us any detailed feedback on the results of individual tasks, such as when they were executed and how long they took to complete.

1.1.1 Workflow as a series of tasks

The challenge of coordinating tasks is hardly a new problem in computing. One of the most common approaches is to define processes in terms of workflows, which represent a collection of tasks to be run, as well as the dependencies between these tasks (i.e., which tasks need to be run first).

Many workflow management systems have been developed over the past decades. Central to most of these systems is the concept of defining tasks as 'units of work' and expressing dependencies between these tasks. This allows workflow management systems to track which tasks should be run when, while also taking care of features such as parallelism, automatic retries of failing jobs, and reporting.

1.1.2 Expressing task dependencies

To determine when to run a given task, a workflow management system needs to know which tasks need to be executed before that task can be run. Such relationships between tasks are typically called task *dependencies*, as in the execution of a task depends on these earlier tasks (its dependencies).

How task dependencies are defined differs between workflow management tools, but definitions typically involve saying which tasks are upstream of the given task (pointing to the tasks dependencies) or saying which tasks are downstream of the task (marking the task as a dependency of other tasks).

One common way of visualizing these dependencies is to depict workflows as

Directed graphs. In this representation, tasks are generally drawn as nodes of the graph, whilst task dependencies are depicted as directed edges between nodes.

Imagine you’re developing a model for an umbrella company who want to predict future umbrella sales across the world given historical sales data and weather predictions. A workflow for such a process could look as follows:

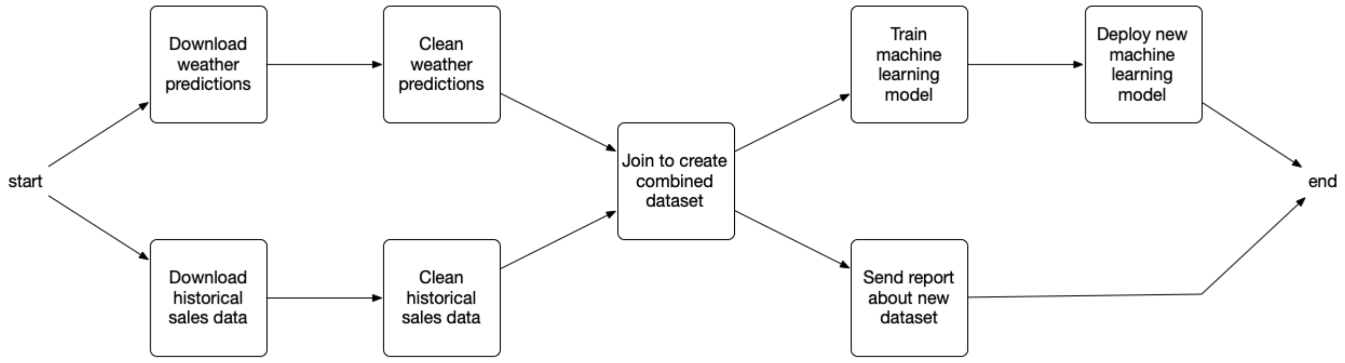

start
Download weather predictions
Clean weather predictions
Download historical sales data
Clean historical sales data
Join to create combined dataset
Train machine learning model
Deploy new machine learning model
Send report about new dataset
end

Figure 1.1 Example of a sequence of tasks executed in a certain order, together forming a workflow. The arrows show ordering between tasks and tell us which tasks must complete before consecutive tasks can start.

1.1.3 Workflow Management Systems

All workflow managers have their own unique way to define workflows. There is no shortage of workflow managers on the market. Several companies/groups encountered similar challenges during the rise of big data and dealing with data workflows, and they developed their interpretation of workflow management systems. While not a complete list, here are some well-known options[1]:

- Make
- Oozie
- Argo
- Kubeflow
- Azkaban (originated from LinkedIn)
- Luigi (originated from Spotify)
- Conductor (originated from Netflix)
- Pinball (originated from Pinterest)
- Airflow (originated from Airbnb)

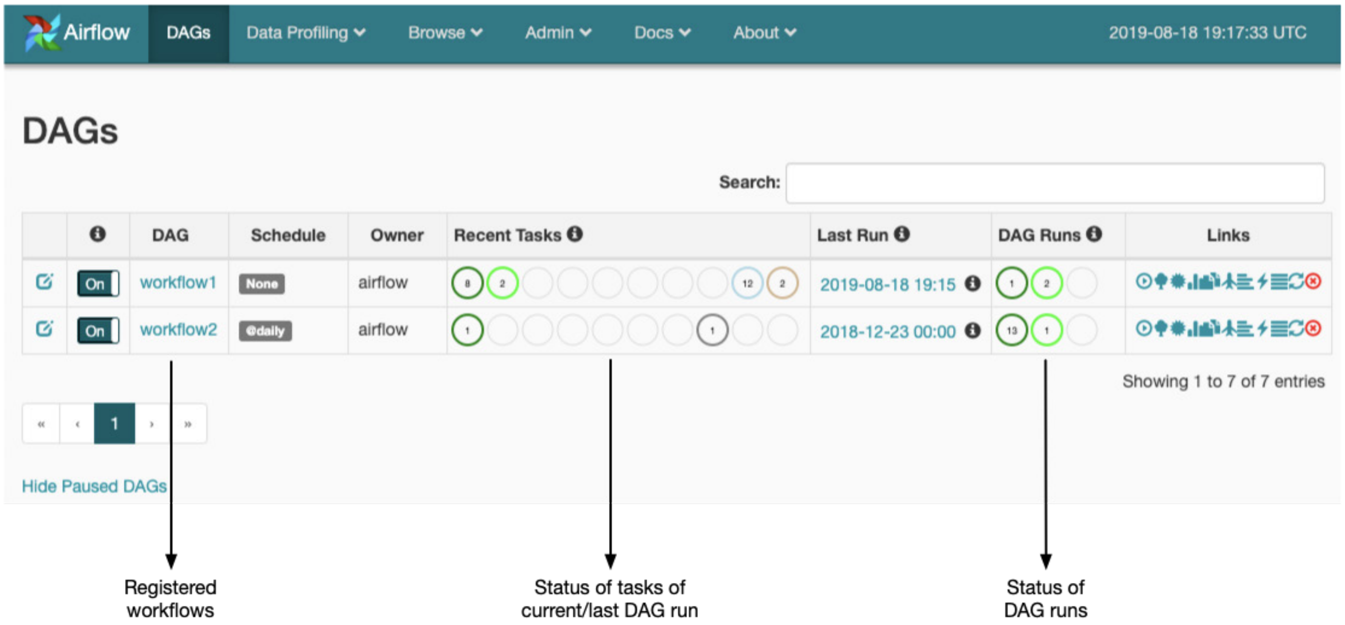
All systems have their strengths and weaknesses. Some are oriented at specific platforms or use cases. In general, some important distinctions between them can be pointed out.

With Oozie, workflows are defined in static XML files. Airflow, on the other hand, allows for dynamic and flexible workflows because they are defined in Python code. Also, Oozie is bound to the Hadoop platform. It has tight integration with Hadoop-specific tools such as Hive and Spark. Shell scripts can also be run, which allows for implicitly running any type of task.

[1] Some tools were originally created by (ex-)employees of a company, however all tools are open sourced and not represented by one single company.

Airflow, on the other hand, is, in essence, a Python framework which allows running any type of task which can be executed by Python, of which Hadoop-tasks is one of the many. The Airflow ecosystem features building blocks for many types of tasks, e.g. sending an email, running a Spark job and running a SQL query on a Postgres database.

Other competitors offering similar "generic" building blocks such as Luigi differ by various features. A few key distinctions between Airflow and Luigi are the availability/lack of a scheduler in Airflow resp. Luigi which allows for scheduling workflows at certain intervals while with Luigi you have to schedule workflows outside the system with, e.g. Cron. Both Airflow and Luigi come with a user interface which shows the status of your workflows, but the Airflow UI allows more operations such as re-running workflows and is generally considered more feature-rich.



2019-08-18 19:17:33 UTC

DAGs

		DAG	Schedule	Owner	Recent Tasks	Last Run	DAG Runs	Links
	On	workflow1	None	airflow	8 2 12 2	2019-08-18 19:15	1 2	
	On	workflow2	@daily	airflow	1 1	2018-12-23 00:00	13 1	

Showing 1 to 7 of 7 entries

Figure 1.2 Airflow UI main page. It shows your workflows and their current status.

Lastly, Airflow is a growing ecosystem of components to run operations on any system. While there are many ways to process your data, our experience as data engineers has taught us that Airflow is a superior platform with an ecosystem of components to operate on various systems.

1.2 An overview of the Airflow components

Airflow consists of various components. The general architecture consists of a webserver, scheduler, and database. The database holds all state of your Airflow system so that you can restart Airflow processes at any time without losing state. Common choices for the database are PostgreSQL and MySQL[2].

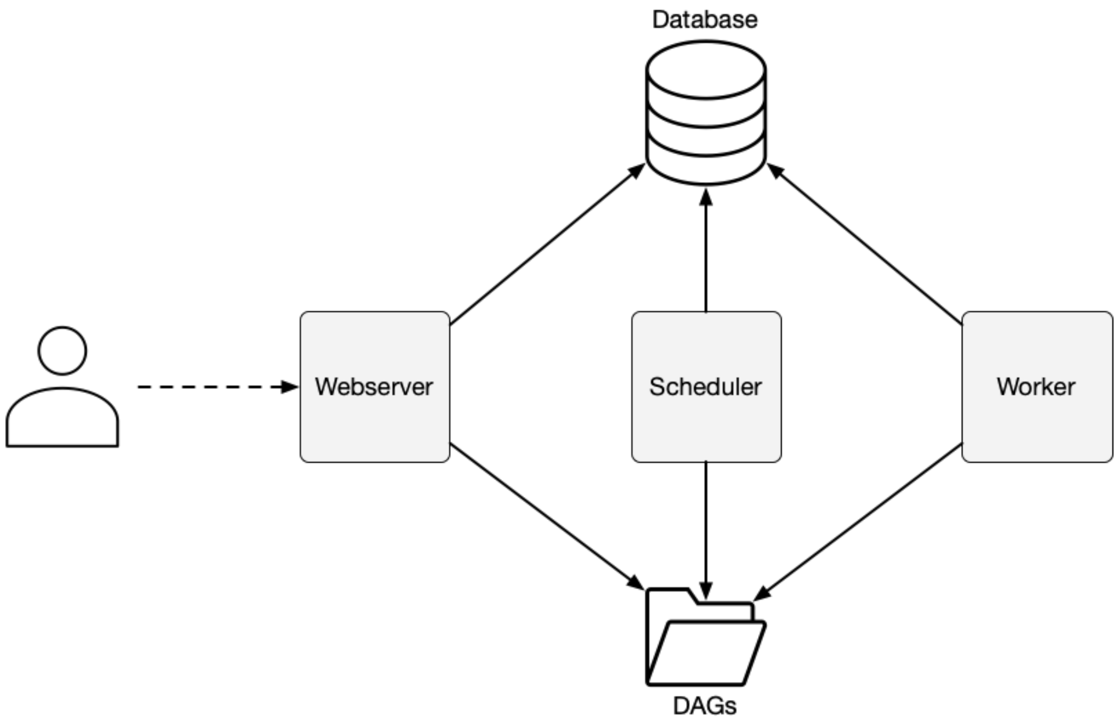


Figure 1.3 General Airflow architecture

The Airflow webserver displays in a web-based user interface the status of your workflows and supports manual operations on your workflows such as running a complete workflow or restarting a single failed task. It displays information both on the workflow level and the individual task level and can displays logs from each task run.

The Airflow scheduler process is responsible for checking whether or not criteria for running tasks are met; are dependent tasks completed successfully, is a task eligible to run given its scheduling interval, are other conditions for execution met, etc. Once all criteria are met, this fact is recorded in the database, and individual tasks are picked up by a worker process to be executed. Depending on your choice of setup this worker process can be a single

[2] Internally Airflow uses SQLAlchemy to communicate with the database. Thus any database supported by SQLAlchemy will be supported by Airflow.

process on a single machine, multiple processes on a single machine or multiple processes distributed across multiple machines. The single process setup is the simplest and easiest for testing or taking a first look at Airflow, but is not advised in production settings because it's slow and does not scale-out.

All Airflow processes require access to a shared location holding the workflows (called *D*irected *A*cyclic *G*raphs or "DAGs"). Airflow repeatedly scans this directory for new files and changes, and stores various properties of workflows in the database. However, on some occasions the workflow files must be re-read and thus must be accessible to all Airflow processes.

1.2.1 Directed Acyclic Graphs

Workflows in Airflow are modeled as *DAG*s: *D*irected *A*cyclic *G*raphs. Airflow is aimed at batch data pipelines where a collection of tasks and dependencies between the tasks together form a graph with a clearly defined start and end. With cycles in the graph, we'd never know when a workflow has finished; hence workflows cannot contain cycles, and hence the acyclic property.

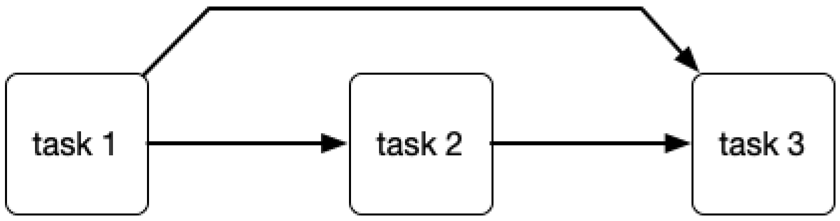


Figure 1.4 Example of a valid DAG. There is only one possible ending state with this structure; after task 3 has finished, the DAG is completed.

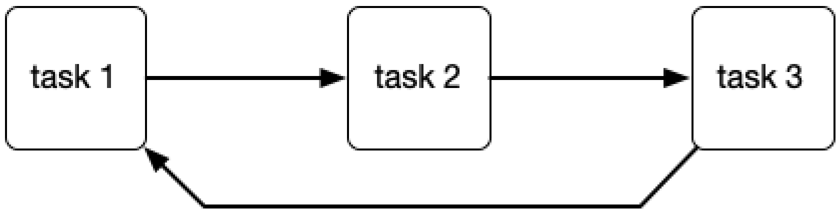


Figure 1.5 Example of an invalid DAG. After task 3 completes, task 1 would be run again and we could never tell when the workflow completes.

A DAG in Airflow is built up from various building blocks. In the next chapters we will unpack and cover the DAG components in detail. DAGs form the core structure around which Airflow structures its workflows. The user interface is a helpful tool to inspect the structure and status of your DAGs.

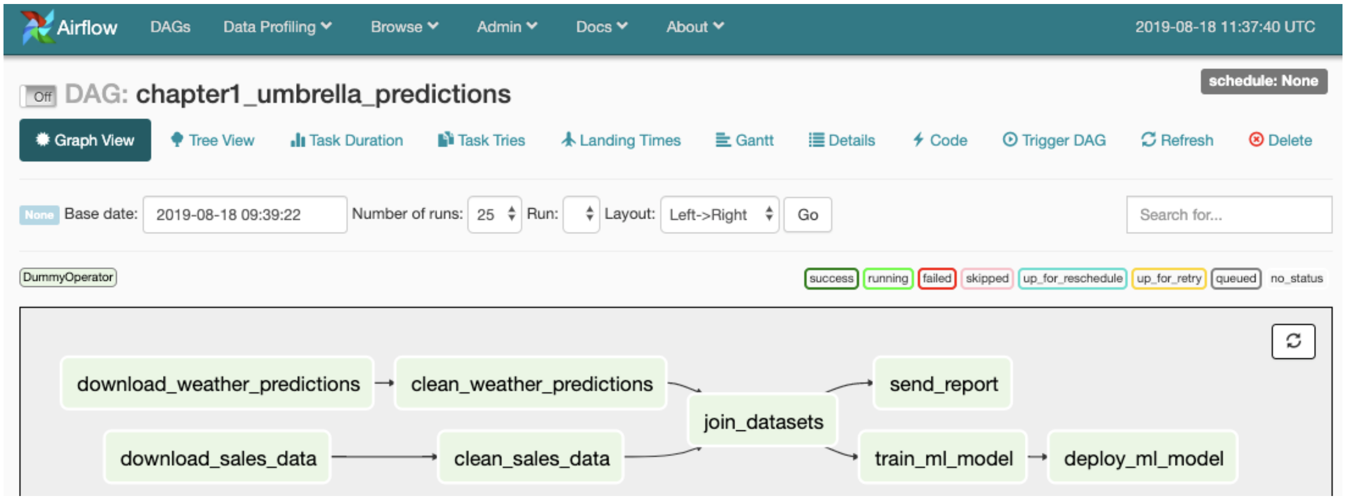


Figure 1.6 Umbrella sales prediction model from Figure 1.1 defined and visualized as a DAG in Airflow.

1.2.2 Batch processing

Airflow operates in the space of batch processes; a series of finite tasks with clearly defined start and end tasks, to run at certain intervals or triggers. Although the concept of workflows also exists in the streaming space, Airflow does not operate there. A framework such as Apache Spark is often used in an Airflow job, triggered by a task in Airflow, to run a given Spark job. When used in combination with Airflow, this is always a Spark batch job and not a Spark streaming job because the batch job is finite and a streaming job could run forever.

1.2.3 Configuration as code

In general, workflow management systems can be divided into two camps: those configured in code (e.g., Python) and those configured in static files (e.g., XML). Python code allows for dynamic and flexible workflows generated using, e.g. for loops and other programming constructs. Static configuration, on the other hand, is less flexible in that sense and typically follows a strict schema.

Configuration as code can be both positive and a negative; the dynamic nature of code can result in a concise piece of code for many tasks. However, there are infinite ways to define code and as such also less strict definitions of workflows.

1.2.4 Scheduling and backfilling

Airflow workflows can be started in various ways: by a manual action, external triggers or by scheduled intervals. In all ways, a series of tasks is run every time the workflow is started. This might work fine for as long as you run your workflow, but at some point in time you might want to change the logic.

For the Umbrella example in Figure 1.1, say you'd like the daily workflow to clean the data differently in the "Clean weather predictions" task. You can change your workflow to produce these new cleaned data and run the new workflow from now on. However, it might be favorable to also run this new logic on all previously completed workflows, on all historical data. This is possible in Airflow with a mechanism called *backfilling* - running workflows back in time. This is only possible of course if external dependencies such as the availability of data can be met. What makes backfilling especially useful is the ability to rerun partial workflows. If the fetching of data is not possible back in time or is a very lengthy process you'd like to avoid, you can rerun partial workflows with backfilling.

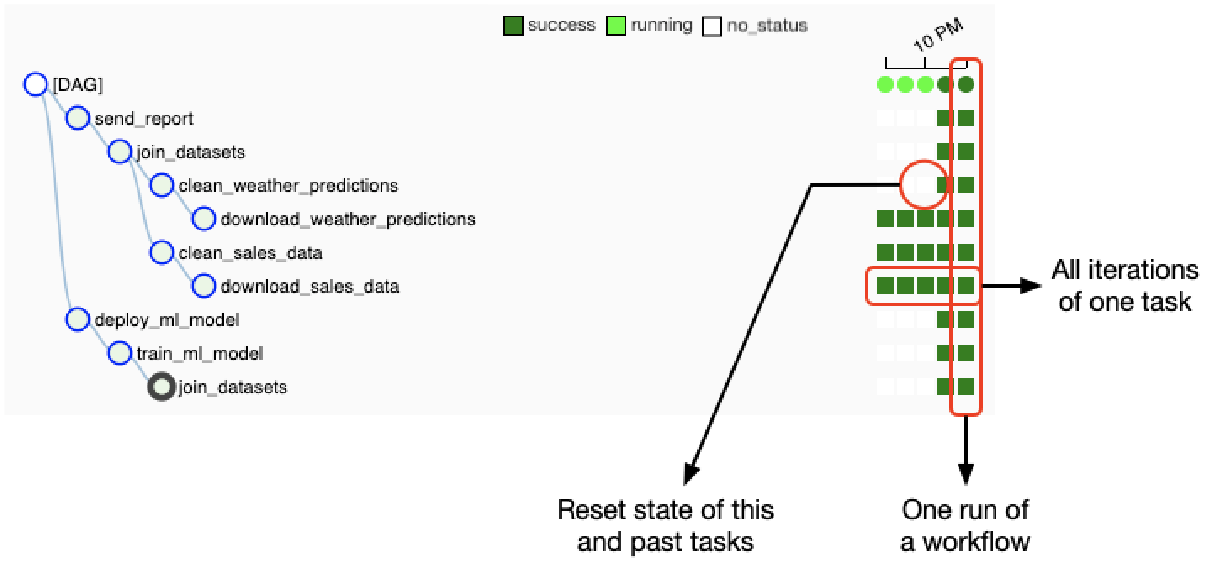


Figure 1.7 Execution of Umbrella sales prediction model from Figure 1.1 displayed over time. The columns show the status of one execution of a workflow, the rows the status of all executions of a single task. The task covered by the red circle, its tasks down the line in the workflow, and its past tasks, were all backfilled with a single operation and in progress to be executed again.

Backfilling preserves all constraints of tasks and runs them, as if time is reversed to that point in time. With each task Airflow provides a runtime context and when rerunning historical tasks, Airflow provides the runtime context as if time was reverted. Besides backfilling, Airflow provides several constructs to manage the lifecycle of tasks and workflows which are covered throughout this book.

1.2.5 Handling failures

In an ideal world all tasks complete successfully. However, the software can fail for virtually any reason and therefore we need ways to automatically deal with such failures, because nobody likes being woken up in the middle of the night. Sometimes failure is acceptable, sometimes it is not. Airflow provides various ways for handling failures, varying from for example stating "it's okay - continue running other tasks," to "retry this one failed task for a maximum of 5 times" to "failure here is unacceptable - fail the entire pipeline." The business logic of the how and when to handle failures can be complex and Airflow can deal with failure on both task and workflow level.

1.3 Choosing Airflow for your workflows

Imagine at your company you are tasked to develop workflows for processing data between multiple systems. A few examples of such tasks could be:

- Performing a daily data dump from a MySQL database into a partitioned Hive table
- Re-running a data pipeline on historical data stored on an FTP server
- Computing reports on yesterday's data and emails the report to, depending on the day of the week, the appropriate person
- Retraining and deploying a data science model based on an hourly data dump
- Waiting for a file to arrive somewhere between 1AM and 5AM and upon arrival immediately process and store it in an AWS S3 bucket

1.3.1 When to not use Airflow?

We mentioned earlier how the Airflow framework allows for flexible and dynamic workflows since DAGs are defined in Python code, as opposed to more static definitions with, e.g. XML or JSON. Both have their benefits, for example XML and JSON can have a schema defined which limits the number of possible ways to define something for the better. Although there are conventions, Python code is not bound by any restrictions leaving a practically unlimited number of ways to define your workflows. This is often considered as positive in contrast to the restrictiveness of XML in terms of readability and conciseness of the code. However it can be regarded a negative since everybody tends to write Python code a little different.

We also mentioned the word "dynamic" several times, which must be clarified. The fact Airflow allows to define workflows in Python code can result in concise workflow definitions. For example, if we have a list of 5 table names, we can create a task to process each table within a for-loop, defining the actual task only once and avoiding code duplication. This is what we refer to with the "dynamicity" of the workflows.

In Airflow there is also the notion of "dynamic DAGs" which refers to the same script resulting in different behavior, i.e. when generating a collection of tasks based on the content of an external file. While the workflow script itself does not change, the external file contents read in the workflow script can change, resulting in different behavior of the workflow. Airflow tends to favor workflows which are not dynamic in structure, meaning workflows whose structure change every time they are run, will not work nicely in Airflow.

1.3.2 Who will find Airflow and this book useful?

All in all, we expect the reader to be a data-savvy person. The data field is ever-growing and as such it's never possible to know everything beforehand. However, we expect you to not be afraid of working with data and programming. Knowing your way with SQL (and at least one programming language) is a must.

The first few chapters of this book aim at covering the basic components of the Airflow ecosystem you must know as a developer. We expect some Python experience, say approximately one year. These chapters should include everything needed to know as a user of the Airflow components. Typical positions with these skills are data analysts and data scientists. We expect you to have basic knowledge of:

- Databases (e.g. have experience with at least relational such as MySQL/Postgres)
- File systems (e.g. reading/writing files from FTP)
- Data types (e.g. know how to mingle with datetimes)
- Python (e.g., know args, kwargs, list comprehensions and templating)
- SQL

If you're interested in extending the Airflow components with your own, this book will also be of interest for you. The typical reader we see here is a data engineer with more Python experience. You have experience with data wrangling in Python, understand the concepts of OOP, data formats (e.g. Parquet/Avro), data structuring (e.g. partitioning/bucketing) and have some experience with the Linux filesystem. The subsequent chapters explain how to build your own Airflow components.

DevOps engineers and systems administrators who are responsible for monitoring data pipelines and maintaining Airflow installations will also find this book beneficial. Best practices for hands-on setting up and maintaining Airflow fill the remaining chapters. We cover logging, monitoring, security, and setting up Airflow in a distributed manner.

1.4 Getting Airflow up and running

In order to get Airflow up and running, you can install Airflow either in your Python environment or run a Docker container. The Docker way is a one-liner:

```
docker run -p 8080:8080 airflowbook/airflow
```

This requires a Docker Engine to be installed on your machine. It will download and run the Airflow Docker container. Once running, you can view Airflow on http://localhost:8080. The second option is to install and run Airflow as a Python package from PyPi:

```
pip install apache-airflow
airflow webserver
```

Again browse to http://localhost:8080, and you will see the first glimpse of Airflow. At this point only the webserver has started, providing a read-only interface. Start the scheduler to have a complete working Airflow instance up and running:

```
airflow scheduler
```

At this point, you can run example workflows, click through the interfaces and view the status of the running and completed tasks. In the next chapter, we get started with developing our own.

1.5 Summary

- Workflow management systems deal with the execution and management of workflows.
- Workflows can be represented as DAGs, in which tasks are depicted by nodes and task dependencies by directed edges between the nodes.
- DAGs have an end state because they cannot contain cycles.
- Workflows in Airflow are modeled as DAGs.
- Tasks in Airflow can run in any language, software or system.
- Airflow enables running tasks and (partial) workflows back in time via backfilling.

2 Anatomy of an Airflow DAG

This chapter covers

- **Implementing the Airflow setup on your own machine**
- **Writing and running your first workflow**
- **Examining the first view at the Airflow interface**
- **Detecting the failures in Airflow**

In the previous chapter, we learned why working with data and the many tools in the data landscape is not an easy task. In this chapter we get started with Airflow and check out an example workflow that uses basic building blocks found in many workflows. It helps to have some Python experience when starting with Airflow; since workflows are defined in Python code, the gap to learning the basics of Airflow is not that big. Getting a workflow up and running with Airflow is often not a hard task; the number of concepts to learn for a newcomer is low. The more complicated part is knowing when to, and when not to make certain choices; something that typically comes with hands-on experience.

2.1 Tracking rocket launches

Rockets are one of mankind's engineering marvels, and every rocket launch attracts attention all around the world. In this chapter we cover the life of a rocket enthusiast named John who tracks and follows every single rocket launch. The news about rocket launches is found in many news sources that John keeps track of, and ideally John would like to have all his rocket news aggregated in a single location. John recently picked up programming and would like to have some sort of automated way to collect information of all rocket launches and eventually some sort of personal insight into the latest rocket news. To start small, John decided to first collect images of rockets.

2.1.1 Launch Library

For the data, we make use of the Launch Library[3] (https://launchlibrary.net), an online repository of data about both historical and future rockets launches from various sources. It is a free and open API, for anybody on the planet. John is currently only interested in upcoming rocket launches. Luckily the Launch Library provides precisely the data he is looking for on this URL: https://launchlibrary.net/1.4/launch?next=5&mode=verbose. It provides data about the next 5 forthcoming rocket launches, together with URLs where to find images of the several rockets. A snippet of the data this URL returns:

```
{
   "launches": [
       {
           "id": 1343,
           "name": "Ariane 5 ECA | Eutelsat 7C & AT&T T-16",
           "windowstart": "June 20, 2019 21:43:00 UTC",
           "windowend": "June 20, 2019 23:30:00 UTC",
           ...
          "rocket": {
               "id": 27,
               "name": "Ariane 5 ECA",
               ...
             "imageURL":
      "https://s3.amazonaws.com/launchlibrary/RocketImages/Ariane+5+ECA_1920.jpg"
           },
           ...
      },
       {
           "id": 1112,
           "name": "Proton-M/Blok DM-03 | Spektr-RG",
           "windowstart": "June 21, 2019 12:17:14 UTC",
           "windowend": "June 21, 2019 12:17:14 UTC",
           ...
          "rocket": {
               "id": 62,
               "name": "Proton-M/Blok DM-03",
               ...
             "imageURL":
      "https://s3.amazonaws.com/launchlibrary/RocketImages/placeholder_1920.png"
           },
           ...
      },
       ...
   ],
   "total": 202,
   "offset": 0,
   "count": 5
}
```

[3] API documentation: https://launchlibrary.net/docs/1.4/api.html

As you can see, the data is in JSON format and provides rocket launch information, and for every launch there's a field "rocket" which contains information about the specific rocket such as id, name and the imageURL. This is exactly what John needs, and he initially draws out the following plan to collect the images:

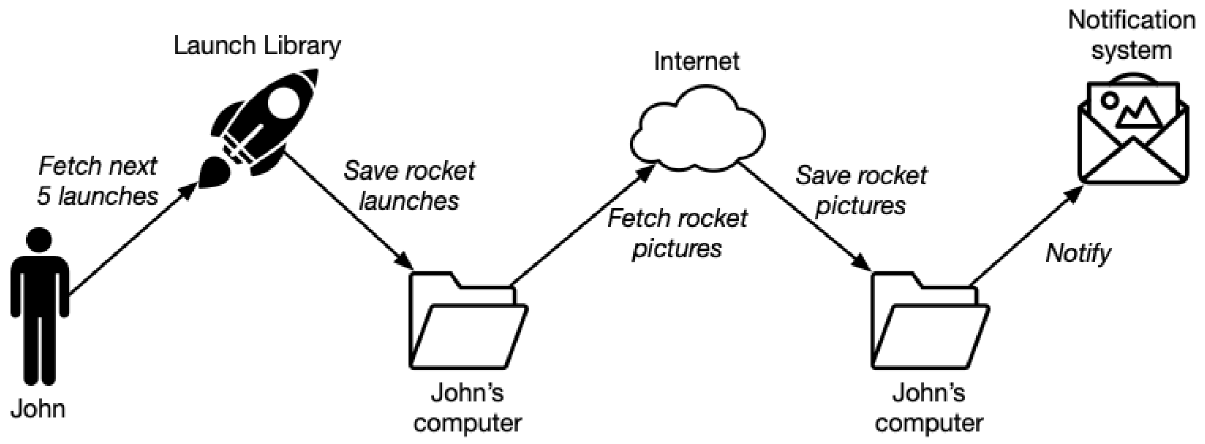


Figure 2.1 John's mental model of downloading rocket pictures

At the end of the day, John's goal is to have a directory filled with rocket images such as this image of the Ariane 5 ECA rocket:

Figure 2.2 Example picture of the Ariane 5 ECA rocket

2.2 Writing your first Airflow DAG

John’s use case is nicely scoped, so let’s check out how to program his plan. It’s only a few steps and in theory, with some Bash-fu, you could work it out in a one-liner. So why would we need a system like Airflow for this job?

The nice thing about Airflow is that we can split a large job, which consists of one or more steps, into individual “tasks” and together form a “DAG.” Each task in a DAG is responsible for a single piece of work. Multiple tasks can be run in parallel, tasks can be restarted at any point in the DAG at any point in time, and tasks can perform different types of work, so we could first run a Bash script and next run a Python script. We worked out John’s use case in an Airflow DAG:

```python
import json
import pathlib

import airflow
import requests
from airflow import DAG
from airflow.operators.bash_operator import BashOperator
from airflow.operators.python_operator import PythonOperator

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

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,
)


def _get_pictures():
   # Ensure directory exists
   pathlib.Path("/tmp/images").mkdir(parents=True, exist_ok=True)

   # Download all pictures in launches.json
   with open("/tmp/launches.json") as f:
       launches = json.load(f)
       image_urls = [launch["rocket"]["imageURL"] for launch in launches["launches"]]
       for image_url in image_urls:
           response = requests.get(image_url)
           image_filename = image_url.split("/")[-1]
           target_file = f"/tmp/images/{image_filename}"
           with open(target_file, "wb") as f:
               f.write(response.content)
           print(f"Downloaded {image_url} to {target_file}")


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

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

download_launches >> get_pictures >> notify
```

We broke down John’s mental model of his workflow into three tasks in Airflow:

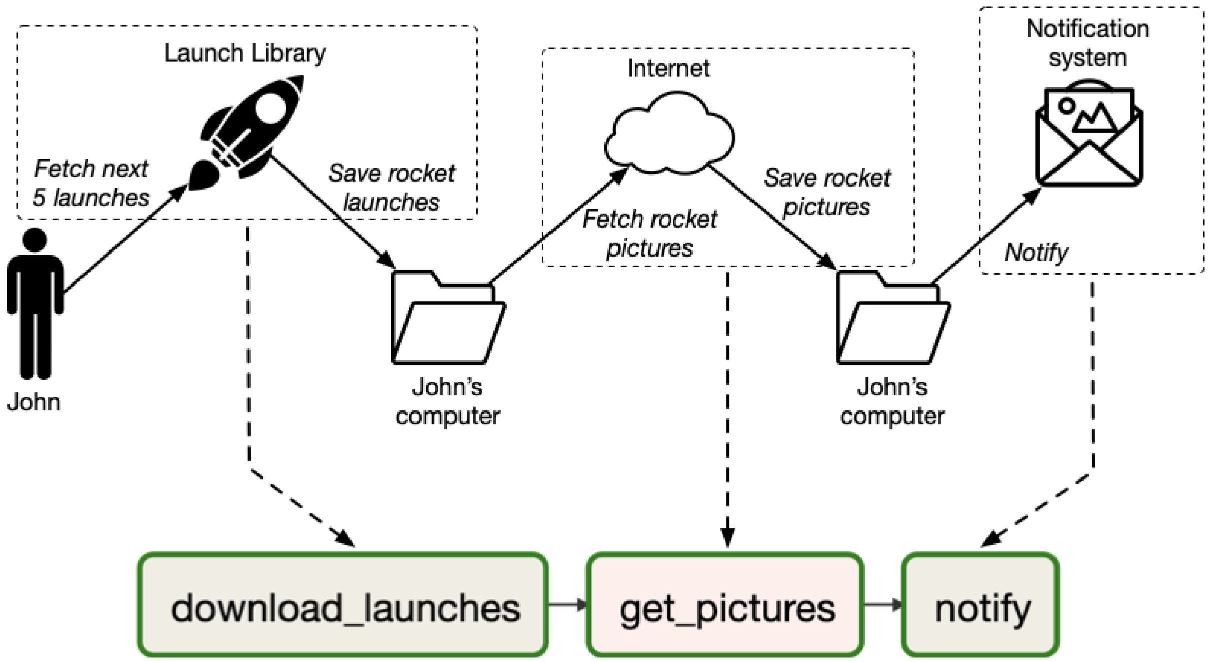


Figure 2.3 John's mental model mapped to tasks in Airflow

Why these three tasks you might ask? Why not download the launches and corresponding pictures in one single task you might wonder? Or why not split up into five tasks? After all, we have five arrows in John's plan? These are all valid questions to ask yourself while developing a workflow, but the truth is there's no right or wrong answer. There are several points to take into consideration, though, and throughout this book we work out many of these use cases to get a feeling for what is right and wrong. Let's first break down the workflow.

Each workflow starts with a DAG object. This is the main starting point of any workflow, without DAG there's no workflow.

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

The DAG is the starting point of any workflow; all tasks within the workflow reference this dag object so that Airflow knows which tasks belong to which DAG. The DAG class takes two required arguments:

1. `dag_id`: The name of the DAG displayed in the Airflow UI
2. `start_date`: The datetime at which the workflow should first start running

Next, an Airflow workflow script consists of one or more operators which perform the actual work. In this case, we can (for example) use the `BashOperator` to run a bash command:

```
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,
)
```

Each operator performs a single unit of work, and multiple operators together form a workflow or DAG in Airflow. Operators run independent of each other, although you can define dependencies. After all, John's workflow wouldn't be useful if you first tried downloading pictures if there is no data about the location of the pictures yet. To make sure the tasks run in the correct order, we can set dependencies between tasks as follows:

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

This ensures the get_pictures task runs only after download_launches has been completed successfully and the notify task only runs after get_pictures has completed successfully.

2.2.1 Tasks vs operators

You might wonder what the difference is between tasks and operators. After all, they both execute a bit of code. In Airflow, operators have a single piece of responsibility: they exist to perform one single piece of work. Some operators perform generic work such as the BashOperator (run a Bash script) and the Python operator (run a Python function), others have more specific use cases such as the EmailOperator (send an email) or the HTTPOperator (call an HTTP endpoint). Either way, they perform a single piece of work.

The role of a DAG is to orchestrate the execution of a collection of operators. That includes the starting & stopping of operators, starting consecutive tasks once an operator is done, ensuring dependencies between operators are met, etc.

In this context and throughout the Airflow documentation, we see the terms "operator" and "task" used interchangeably. From a user's perspective, they refer to the same thing, and the two often substitute each other in discussions. Operators provide the actual implementation and Airflow has a class called BaseOperator and many subclasses inheriting from the BaseOperator such as the BashOperator and PythonOperator.

There is a difference, though. Tasks in Airflow manage the execution of an Operator; they can be thought of as a small "wrapper" or "manager" around an operator which ensures the operator executes correctly. The user can focus on the work to be done by using and creating operators, while Airflow ensures the proper execution of work via tasks:

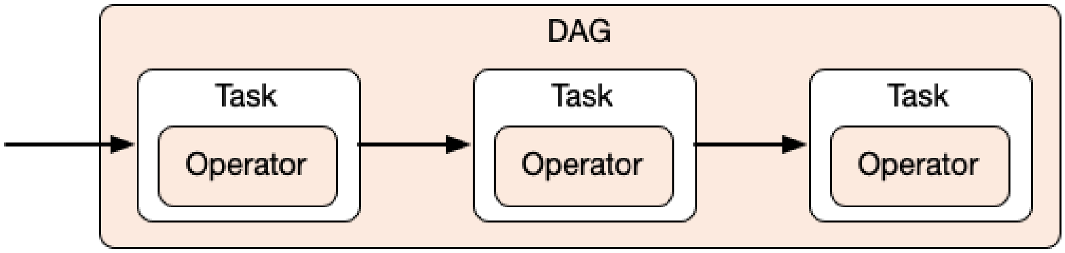


Figure 2.4 DAG and Operators are used by Airflow users, tasks are internal elements to manage operator state.

2.2.2 Running arbitrary Python code

Fetching the data for the next 5 rocket launches was a single curl command in Bash, which is easily executed with the BashOperator. However parsing the JSON result, selecting the image URLs from it and downloading the respective images requires a bit more effort. Although all this is still possible in a Bash one-liner, it is often easier and more readable with a few lines of Python or any other language of your choice. Since Airflow code is defined in Python, it is very convenient to keep both the workflow and execution logic in the same script. For downloading the rocket pictures we implemented the following:

```python
def _get_pictures():
   # Ensure directory exists
   pathlib.Path("/tmp/images").mkdir(parents=True, exist_ok=True)

   # Download all pictures in launches.json
   with open("/tmp/launches.json") as f:
       launches = json.load(f)
       image_urls = [launch["rocket"]["imageURL"] for launch in launches["launches"]]
       for image_url in image_urls:
           response = requests.get(image_url)
           image_filename = image_url.split("/")[-1]
           target_file = f"/tmp/images/{image_filename}"
           with open(target_file, "wb") as f:
               f.write(response.content)
           print(f"Downloaded {image_url} to {target_file}")


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

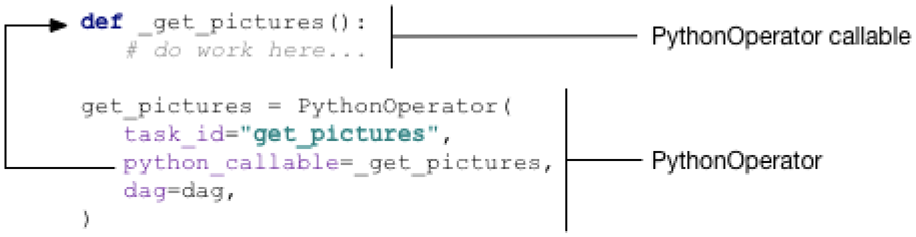
The PythonOperator in Airflow is responsible for running any Python code. Just like the BashOperator used before, this and all other operators require a `task_id`. The `task_id` is referenced when running a task and displayed in the UI.

The use of a PythonOperator is always twofold:

1. We define the operator itself (`get_pictures`) and

2. The `python_callable` argument points to a callable, typically a function (`_get_pictures`)

When running the operator, the Python function is called and will execute the function. Let’s break it down. The basic usage of the PythonOperator always looks as follows:



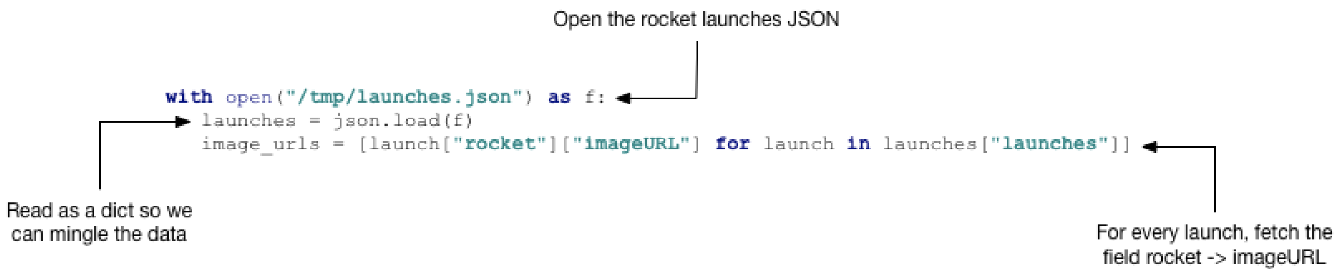
```
def _get_pictures():
    # do work here...

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

Although not required, for convenience we keep the variable name “get_pictures” equal to the task_id.

```
# Ensure directory exists
pathlib.Path("/tmp/images").mkdir(parents=True, exist_ok=True)
```

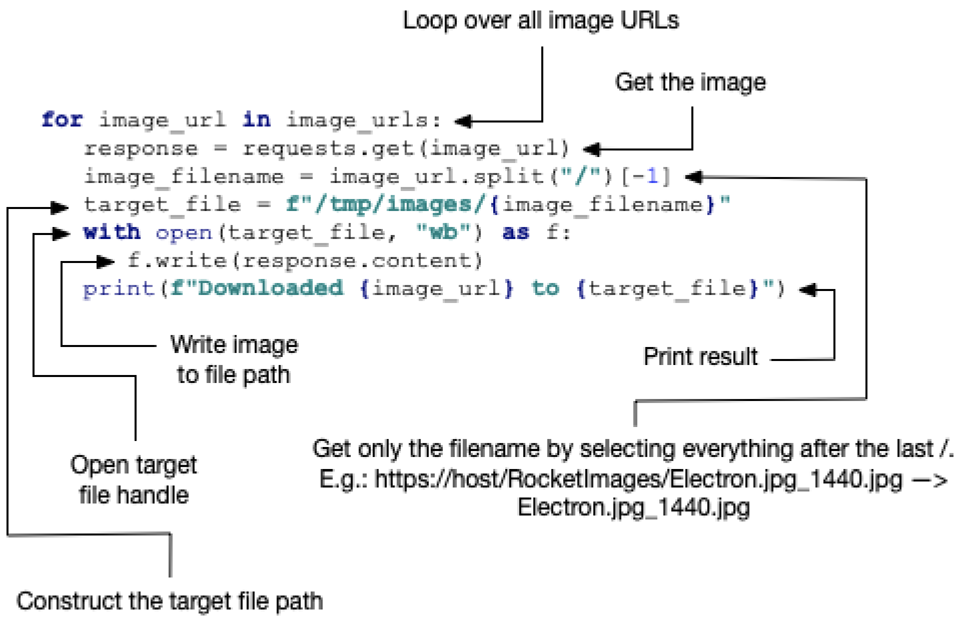
First step in the callable is to ensure the directory in which the images will be stored exists. Next, we open the result downloaded from the Launch Library API and extract the image URLs for every launch:



```
with open("/tmp/launches.json") as f:
    launches = json.load(f)
    image_urls = [launch["rocket"]["imageURL"] for launch in launches["launches"]]
```

Open the rocket launches JSON
Read as a dict so we can mingle the data
For every launch, fetch the field rocket -> imageURL

Each image URL is called to download the image and save it in /tmp/images:


Loop over all image URLs
Get the image
Write image to file path
Print result
Open target file handle
Get only the filename by selecting everything after the last /. E.g.: https://host/RocketImages/Electron.jpg_1440.jpg —> Electron.jpg_1440.jpg
Construct the target file path

```
for image_url in image_urls:
    response = requests.get(image_url)
    image_filename = image_url.split("/")[-1]
    target_file = f"/tmp/images/{image_filename}"
    with open(target_file, "wb") as f:
        f.write(response.content)
    print(f"Downloaded {image_url} to {target_file}")
```

2.3 Running a DAG in Airflow

Now we have our basic rocket launch DAG, let's get it up and running and view it in the Airflow UI. The bare minimum Airflow consists of two core components: (1) a scheduler and (2) a webserver. To get Airflow up and running, you can install Airflow either in your Python environment or run a Docker container. The Docker way is a one-liner:

```
docker run -p 8080:8080 airflowbook/airflow
```

This requires a Docker Engine to be installed on your machine. It will download and run the Airflow Docker container. Once running, you can view Airflow on http://localhost:8080. The second option is to install and run Airflow as a Python package from PyPi:

```
pip install apache-airflow
```

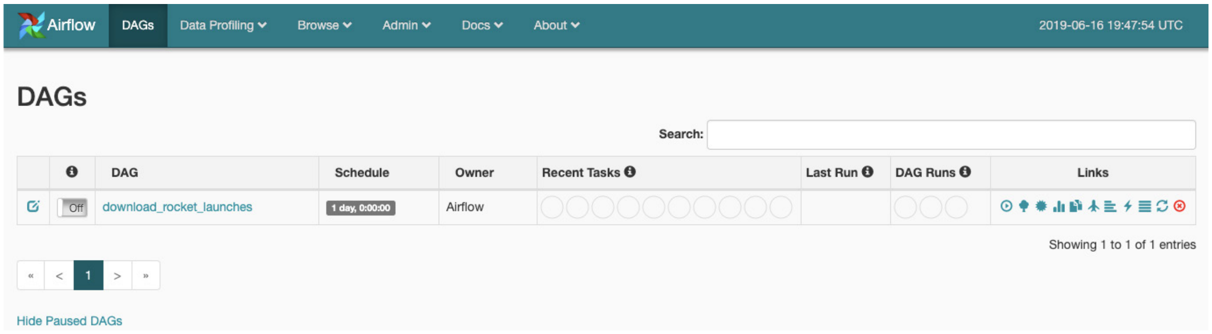
Make sure you install `apache-airflow` and not just `airflow`. Together with joining the Apache Foundation in 2016, the PyPi `airflow` repository was renamed to `apache-airflow`. Since many people were still installing `airflow`, instead of removing the old repository, it was kept as a dummy to provide everybody a message pointing to the correct repository.

Now that you've installed Airflow, start it by initializing the metastore (a database in which all Airflow state is stored), copying the rocket launch DAG into the DAGs directory, and starting the scheduler and webserver:

```
airflow initdb
cp download_rocket_launches.py ~/airflow/dags/
```

```
airflow scheduler
airflow webserver
```

Note the scheduler and webserver are both continuous processes which keep your terminal open, so run either in the background with `airflow scheduler &` or open a second terminal window to run the scheduler and webserver separately. After you're set up, browse to http://localhost:8080 to view Airflow.



		DAG	Schedule	Owner	Recent Tasks	Last Run	DAG Runs	Links
	Off	download_rocket_launches	1 day, 0:00:00	Airflow				

Figure 2.5 Airflow home screen

This is the first glimpse of Airflow you will see. Currently the only DAG is the download_rocket_launches which is available to Airflow in the DAGs directory. There's a lot of information on the main view, but let's inspect the download_rocket_launches DAG first. Click on the DAG name to open it and inspect the so-called Graph View:

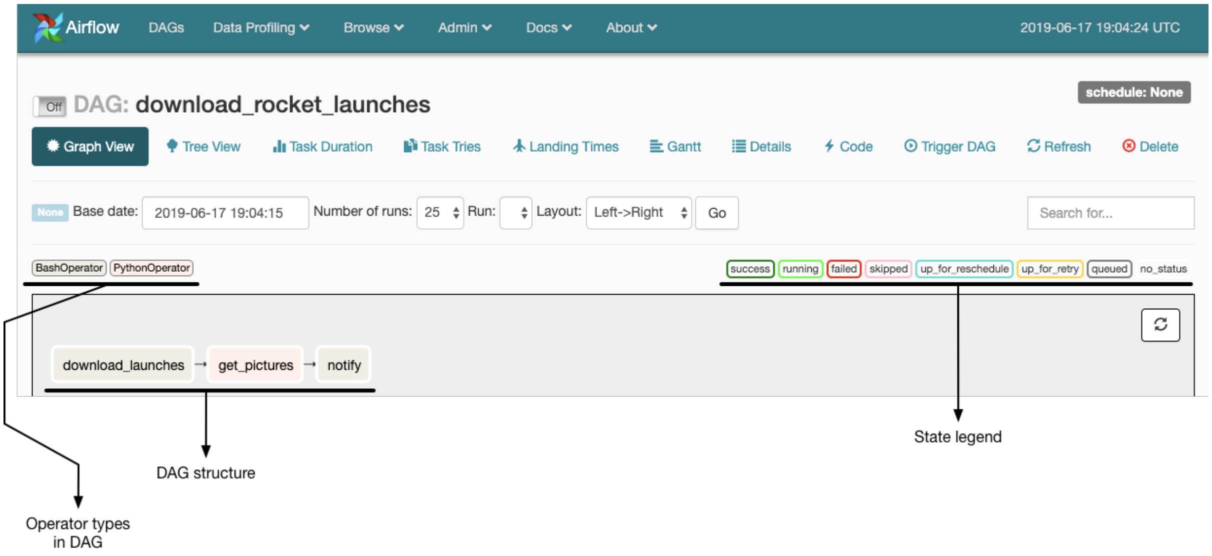


Figure 2.6 Airflow Graph View

This view shows us the structure of the DAG script provided to Airflow. Once placed in the DAGs directory, Airflow will read the script and pull out the bits and pieces that together form a DAG, so it can be visualized in the UI. The graph view shows us the structure of the DAG, how and in which order all tasks in the DAG are connected and will be run. This is one of the views you will probably use the most while developing your workflows.

The state legend shows all colors you might see when running, so let's see what happens and run the DAG. First, the DAG requires to be "On" to be run, toggle the "Off" button for that. Next, click on "Trigger DAG" to run it.

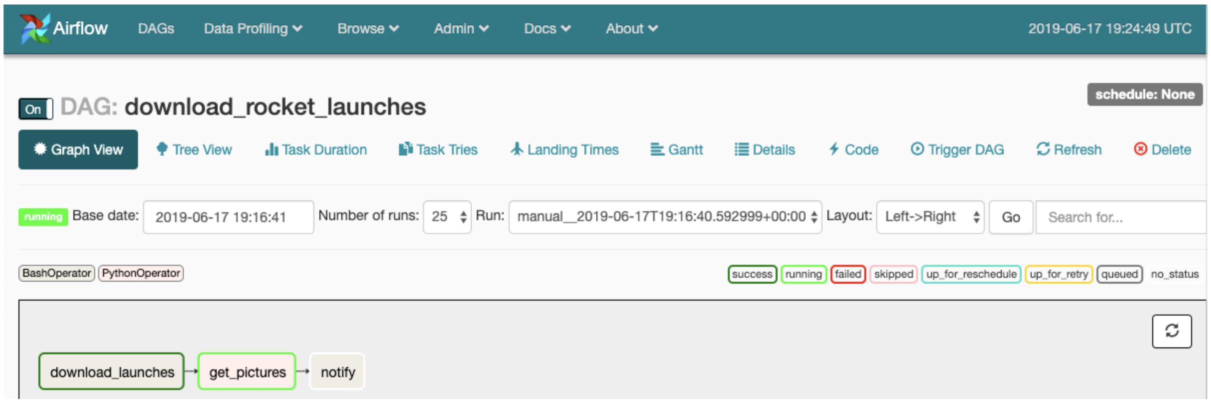


Figure 2.7 Graph View displaying a running DAG

After triggering the DAG, it will start running and you will see the current state of the workflow represented by colors. Since we set dependencies between our tasks, consecutive tasks only start running once the previous tasks have been completed. Let's check the result of the "notify" task. In a real use case, you probably want to send an email or e.g. Slack notification to inform about the new images. For sake of simplicity, it now prints the number of downloaded images. Let's check the logs.

All task logs are collected in Airflow so we can search in the UI for output or potential issues in case of failure. Click on a completed "notify" task and you will see a pop-up with several options:

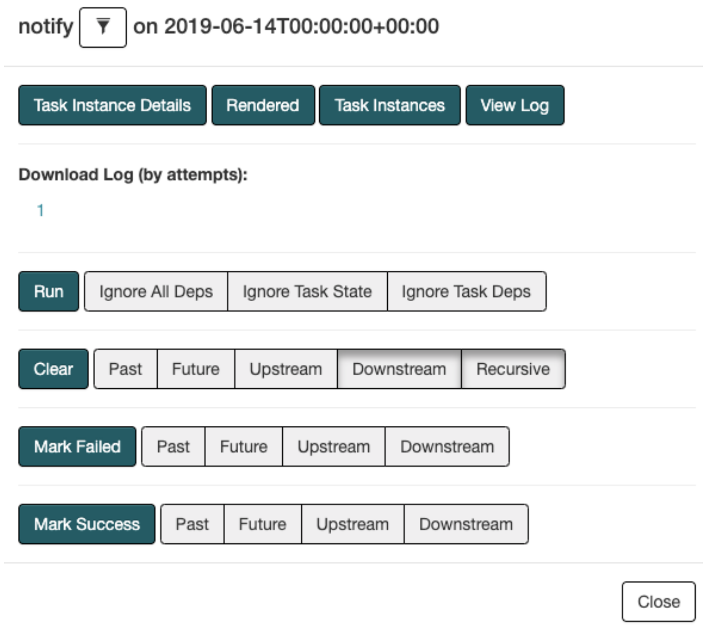
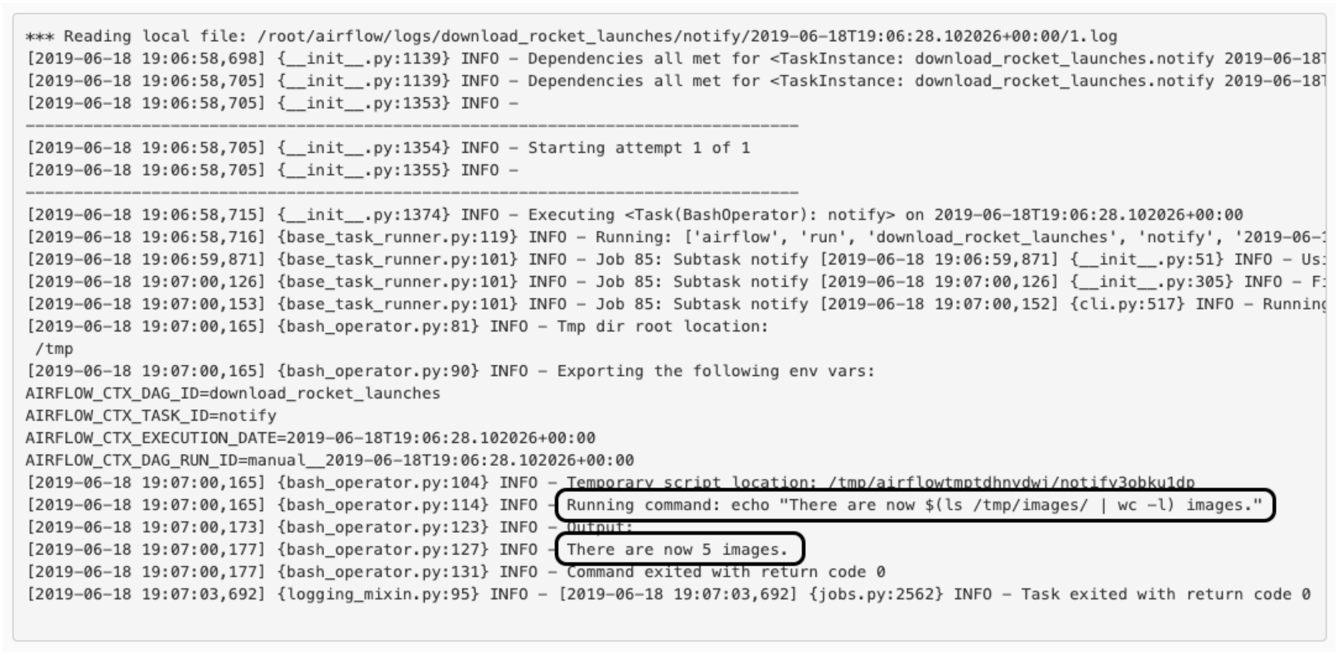


Figure 2.8 Task pop up options

Click on the top-right button “View Log” to inspect the logs:



```
*** Reading local file: /root/airflow/logs/download_rocket_launches/notify/2019-06-18T19:06:28.102026+00:00/1.log
[2019-06-18 19:06:58,698] {__init__.py:1139} INFO - Dependencies all met for <TaskInstance: download_rocket_launches.notify 2019-06-18T
[2019-06-18 19:06:58,705] {__init__.py:1139} INFO - Dependencies all met for <TaskInstance: download_rocket_launches.notify 2019-06-18T
[2019-06-18 19:06:58,705] {__init__.py:1353} INFO -
--------------------------------------------------------------------------------
[2019-06-18 19:06:58,705] {__init__.py:1354} INFO - Starting attempt 1 of 1
[2019-06-18 19:06:58,705] {__init__.py:1355} INFO -
--------------------------------------------------------------------------------
[2019-06-18 19:06:58,715] {__init__.py:1374} INFO - Executing <Task(BashOperator): notify> on 2019-06-18T19:06:28.102026+00:00
[2019-06-18 19:06:58,716] {base_task_runner.py:119} INFO - Running: ['airflow', 'run', 'download_rocket_launches', 'notify', '2019-06-1
[2019-06-18 19:06:59,871] {base_task_runner.py:101} INFO - Job 85: Subtask notify [2019-06-18 19:06:59,871] {__init__.py:51} INFO - Us
[2019-06-18 19:07:00,126] {base_task_runner.py:101} INFO - Job 85: Subtask notify [2019-06-18 19:07:00,126] {__init__.py:305} INFO - F
[2019-06-18 19:07:00,153] {base_task_runner.py:101} INFO - Job 85: Subtask notify [2019-06-18 19:07:00,152] {cli.py:517} INFO - Running
[2019-06-18 19:07:00,165] {bash_operator.py:81} INFO - Tmp dir root location:
 /tmp
[2019-06-18 19:07:00,165] {bash_operator.py:90} INFO - Exporting the following env vars:
AIRFLOW_CTX_DAG_ID=download_rocket_launches
AIRFLOW_CTX_TASK_ID=notify
AIRFLOW_CTX_EXECUTION_DATE=2019-06-18T19:06:28.102026+00:00
AIRFLOW_CTX_DAG_RUN_ID=manual__2019-06-18T19:06:28.102026+00:00
[2019-06-18 19:07:00,165] {bash_operator.py:104} INFO - Temporary script location: /tmp/airflowtmptdhnvdwi/notify3obku1dp
[2019-06-18 19:07:00,165] {bash_operator.py:114} INFO - Running command: echo "There are now $(ls /tmp/images/ | wc -l) images."
[2019-06-18 19:07:00,173] {bash_operator.py:123} INFO - Output:
[2019-06-18 19:07:00,177] {bash_operator.py:127} INFO - There are now 5 images.
[2019-06-18 19:07:00,177] {bash_operator.py:131} INFO - Command exited with return code 0
[2019-06-18 19:07:03,692] {logging_mixin.py:95} INFO - [2019-06-18 19:07:03,692] {jobs.py:2562} INFO - Task exited with return code 0
```

Figure 2.9 Print statement displayed in logs

The logs are quite verbose by default but display the number of downloaded images in the outlog. Finally, we can open the /tmp/images directory and view them:

Figure 2.10 Resulting rocket pictures

2.4 Running at regular intervals

Rocket enthusiast John is happy now that he has a workflow up and running in Airflow, which he can trigger every now and then to collect the latest rocket pictures. He can see the status of his workflow in the Airflow UI, which is already an improvement compared to a script on the command line he was running before. But he still needs to trigger his workflow by hand every now and then which should be automated. After all, nobody likes doing repetitive tasks which computers are good at.

In Airflow, we can schedule a DAG to run at certain intervals, e.g., once an hour, day or month. This is controlled on the DAG by setting the schedule_interval argument:

```
dag = DAG(
   dag_id="download_rocket_launches",
   start_date=airflow.utils.dates.days_ago(14),
   schedule_interval="@daily",
)
```

Setting the schedule_interval to "@daily" tells Airflow to run this workflow once a day, so that John doesn't have to trigger it manually once a day. The behaviour of this is best viewed in the Tree View:

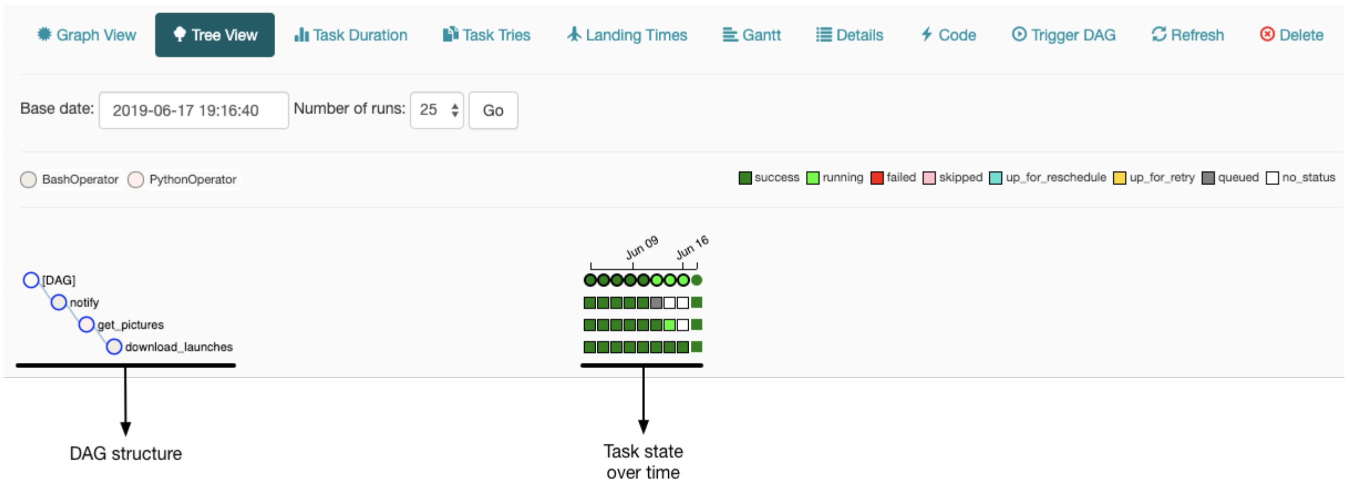


Figure 2.11 Airflow Tree View

The tree view is similar to the graph view but displays the graph structure as is runs overtime. An overview of the status of all runs of a single workflow can seen here.

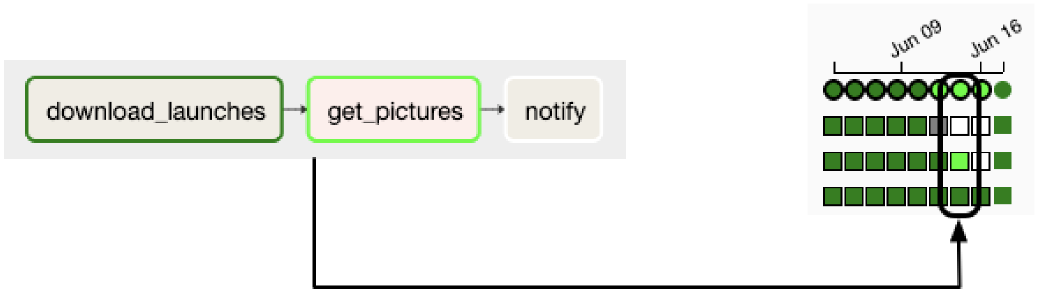


Figure 2.12 Relationship between Graph and Tree View

The structure of the DAG is displayed to fit a "rows and columns" layout, specifically the status of all runs of the specific DAG, where each column represents a single run at some point in time.

When we set the schedule_interval to "@daily," Airflow knew it had to run this DAG once a day. Given the start_date provided to the DAG of 14 days ago, that means the time from 14 days ago up to now can be divided into 14 equal intervals of 1 day. Since both the start and end datetime of these 14 intervals lie in the past, they will start running once we provide a schedule_interval to Airflow. The semantics of the schedule interval and various ways to configure it are covered in more detail in chapter 3.

2.5 Handling failing tasks

So far, we've seen only green in the Airflow UI. But what happens if something fails? It's not uncommon for tasks to fail, which could be for a multitude of reasons, e.g. an external service being down, network connectivity issues or a broken disk. Say at some point we experienced a network hiccup while getting the pictures, we would see a failing task in Airflow, which would look as follows:

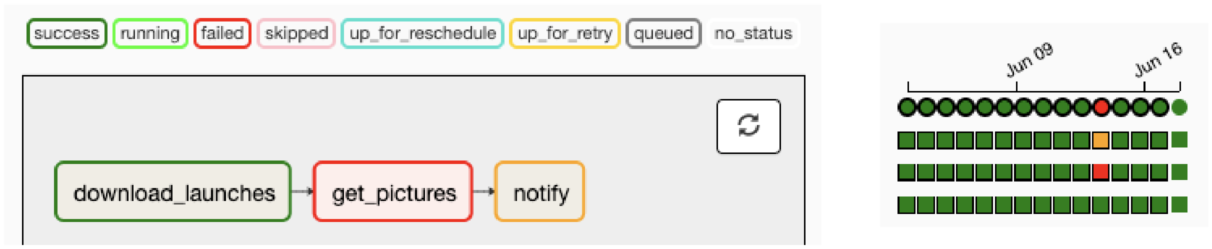
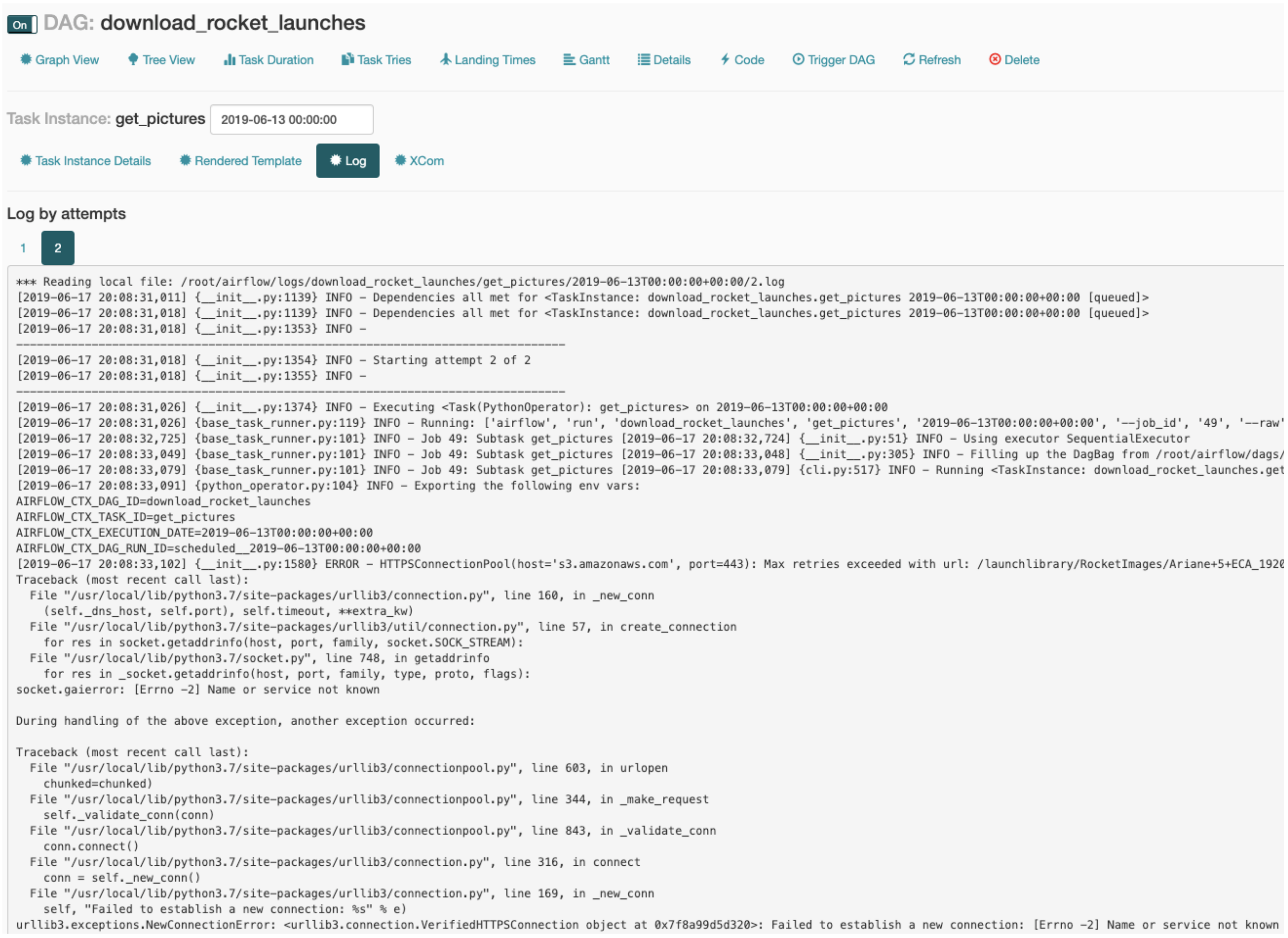


Figure 2.13 Failure displayed in Graph and Tree View

The specific failed task would be displayed in red in both the graph and tree views as a result of not being able to get the images from the internet and therefore raise an error. The successive “notify” task would not run at all because it’s dependent on the successful state of the “get_pictures” task. Such task instances are displayed in orange. By default all previous tasks must run successfully and any successive task of a failed task will not run.

Let’s figure out the issue by inspecting the logs again. Open the logs of the “get_pictures” task:



```
*** Reading local file: /root/airflow/logs/download_rocket_launches/get_pictures/2019-06-13T00:00:00+00:00/2.log
[2019-06-17 20:08:31,011] {__init__.py:1139} INFO - Dependencies all met for <TaskInstance: download_rocket_launches.get_pictures 2019-06-13T00:00:00+00:00 [queued]>
[2019-06-17 20:08:31,018] {__init__.py:1139} INFO - Dependencies all met for <TaskInstance: download_rocket_launches.get_pictures 2019-06-13T00:00:00+00:00 [queued]>
[2019-06-17 20:08:31,018] {__init__.py:1353} INFO -
--------------------------------------------------------------------------------
[2019-06-17 20:08:31,018] {__init__.py:1354} INFO - Starting attempt 2 of 2
[2019-06-17 20:08:31,018] {__init__.py:1355} INFO -
--------------------------------------------------------------------------------
[2019-06-17 20:08:31,026] {__init__.py:1374} INFO - Executing <Task(PythonOperator): get_pictures> on 2019-06-13T00:00:00+00:00
[2019-06-17 20:08:31,026] {base_task_runner.py:119} INFO - Running: ['airflow', 'run', 'download_rocket_launches', 'get_pictures', '2019-06-13T00:00:00+00:00', '--job_id', '49', '--raw'
[2019-06-17 20:08:32,725] {base_task_runner.py:101} INFO - Job 49: Subtask get_pictures [2019-06-17 20:08:32,724] {__init__.py:51} INFO - Using executor SequentialExecutor
[2019-06-17 20:08:33,049] {base_task_runner.py:101} INFO - Job 49: Subtask get_pictures [2019-06-17 20:08:33,048] {__init__.py:305} INFO - Filling up the DagBag from /root/airflow/dags/
[2019-06-17 20:08:33,079] {base_task_runner.py:101} INFO - Job 49: Subtask get_pictures [2019-06-17 20:08:33,079] {cli.py:517} INFO - Running <TaskInstance: download_rocket_launches.get
[2019-06-17 20:08:33,091] {python_operator.py:104} INFO - Exporting the following env vars:
AIRFLOW_CTX_DAG_ID=download_rocket_launches
AIRFLOW_CTX_TASK_ID=get_pictures
AIRFLOW_CTX_EXECUTION_DATE=2019-06-13T00:00:00+00:00
AIRFLOW_CTX_DAG_RUN_ID=scheduled__2019-06-13T00:00:00+00:00
[2019-06-17 20:08:33,102] {__init__.py:1580} ERROR - HTTPSConnectionPool(host='s3.amazonaws.com', port=443): Max retries exceeded with url: /launchlibrary/RocketImages/Ariane+5+ECA_1920
Traceback (most recent call last):
  File "/usr/local/lib/python3.7/site-packages/urllib3/connection.py", line 160, in _new_conn
    (self._dns_host, self.port), self.timeout, **extra_kw)
  File "/usr/local/lib/python3.7/site-packages/urllib3/util/connection.py", line 57, in create_connection
    for res in socket.getaddrinfo(host, port, family, socket.SOCK_STREAM):
  File "/usr/local/lib/python3.7/socket.py", line 748, in getaddrinfo
    for res in _socket.getaddrinfo(host, port, family, type, proto, flags):
socket.gaierror: [Errno -2] Name or service not known

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "/usr/local/lib/python3.7/site-packages/urllib3/connectionpool.py", line 603, in urlopen
    chunked=chunked)
  File "/usr/local/lib/python3.7/site-packages/urllib3/connectionpool.py", line 344, in _make_request
    self._validate_conn(conn)
  File "/usr/local/lib/python3.7/site-packages/urllib3/connectionpool.py", line 843, in _validate_conn
    conn.connect()
  File "/usr/local/lib/python3.7/site-packages/urllib3/connection.py", line 316, in connect
    conn = self._new_conn()
  File "/usr/local/lib/python3.7/site-packages/urllib3/connection.py", line 169, in _new_conn
    self, "Failed to establish a new connection: %s" % e)
urllib3.exceptions.NewConnectionError: <urllib3.connection.VerifiedHTTPSConnection object at 0x7f8a99d5d320>: Failed to establish a new connection: [Errno -2] Name or service not known
```

Figure 2.14 Stack trace of failed `get_pictures` task

In the stack traces, we uncover the potential cause of the issue:

```
urllib3.exceptions.NewConnectionError: <urllib3.connection.VerifiedHTTPSConnection
      object at 0x7f8a99d5d320>: Failed to establish a new connection: [Errno -2]
      Name or service not known
```

This indicates urllib3 is trying to establish a connection but cannot, which could hint at a firewall rule blocking the connecting or no internet connectivity. Now that we found the issue and assume to have fixed the issue, let's restart the task. It would be unnecessary to restart the entire workflow, and a nice feature of Airflow is that you can restart from the point of failure and onwards, without having to restart any previously succeeded tasks.

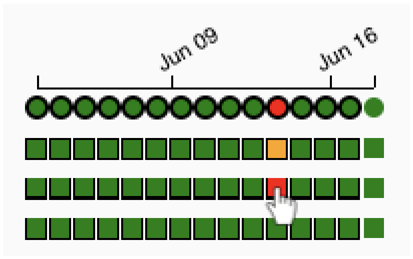
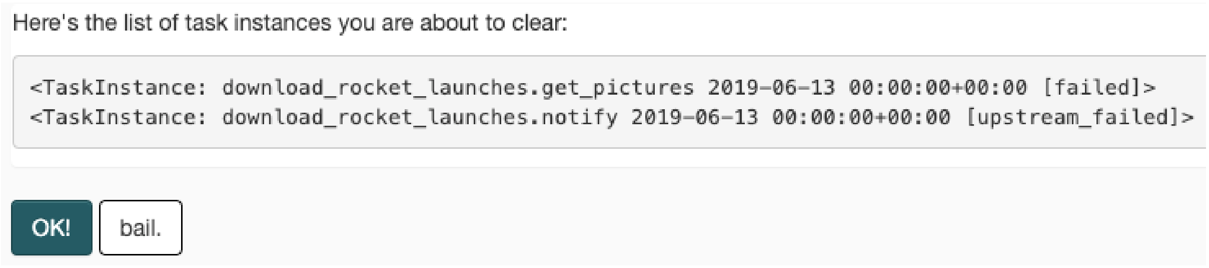


Figure 2.15 Click on a failed task for options to clear it

Click on the failed task, and now click the "Clear" button in the pop up. It will show you the tasks you're about to clear; meaning you will "reset" the state of these tasks and Airflow will rerun them:


Here's the list of task instances you are about to clear:

```
<TaskInstance: download_rocket_launches.get_pictures 2019-06-13 00:00:00+00:00 [failed]>
<TaskInstance: download_rocket_launches.notify 2019-06-13 00:00:00+00:00 [upstream_failed]>
```

Figure 2.16 Clearing the state of get_pictures and successive tasks

Click "OK!" and the failed task and its successive tasks will be cleared:

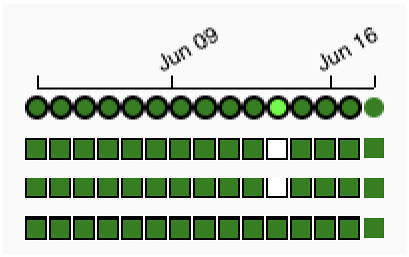


Figure 2.17 Cleared tasks displayed in Graph View

If the internet connectivity issues are resolved, the tasks will now run successfully and make the whole tree view green:

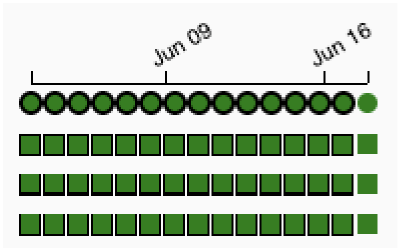


Figure 2.18 Successfully completed tasks after clearing failed tasks

In any piece of software, there are many reasons for failure. In Airflow workflows, sometimes failure is accepted, sometimes it is not, and sometimes it is only in certain conditions. The criteria for dealing while failure can be configured on any level in the workflow, and is covered in more detail in chapter 4.

2.6 Summary

- Workflows in Airflow are represented in DAGs.
- Operators represent a single unit of work.
- Airflow contains an array of operators both for generic and specific types of work.
- The Airflow UI offers a graph view for viewing the DAG structure, and tree view for viewing DAG runs overtime.
- Failed tasks can be restarted anywhere in the DAG.

3
Scheduling in Airflow

This chapter covers

- **Running DAGs regularly using schedule intervals**
- **Constructing efficient DAGs that load and process data incrementally**
- **Designing your DAGs for re-processing past datasets using backfilling**

In the previous chapter, we explored Airflow's UI and showed you how to define a basic Airflow DAG and run this DAG every day by defining a schedule interval. In this chapter, we will dive a bit deeper into the concept of scheduling in Airflow and explore how this allows you to process data incrementally at regular intervals. First, we'll introduce a small use case focussed on analyzing user events from our website and explore how we can build a DAG to analyze these events at regular intervals. Next, we'll explore ways to make this process more efficient by taking an incremental approach to analyzing our data and how this ties into Airflow's concept of execution dates. Finally, we'll finish by showing how we can fill in past gaps in our dataset using backfilling and discussing some important properties of proper Airflow tasks.

3.1 Running tasks at regular intervals

As we've already seen in Chapter 2, Airflow DAGs can be run at regular intervals by defining a schedule interval for the DAG. Schedule intervals can be defined using the *schedule_interval* argument when constructing the DAG. By default, the value of this argument is None, which means that the DAG will not be scheduled and will only be run when triggered manually from the UI or the API. In this section, we will explore several different types of schedule intervals and examine how these affect the scheduling of your DAG.

3.1.1 Use case: processing user events

To understand how Airflow's scheduling works, we'll first consider a small example. Imagine we have a service that tracks user behaviour on our website and allows us to see which pages users (identified by an IP address) accessed on our website. For marketing purposes, we would like to know how many different pages are accessed by our users and how much time they spend during each visit. To get an idea of how this behaviour changes over time, we want to calculate these statistics on a daily basis as this allows to compare changes across different days and larger time periods.

For practical reasons, the external tracking service does not store data for more than 30 days. This means that we need to store and accumulate this data ourselves, as we want to retain our history for longer periods of time. Normally, because the raw data might be quite large, it would make sense to store this data in a cloud storage service such as Amazon's S3 or Google's Cloud Storage service, as these services combine high durability with relatively low costs. However, for simplicity's sake, we won't worry about these things yet and keep our data locally.

To simulate this example, we have created a simple (local) API that allows us to retrieve user events. For example, we can retrieve the full list of available events from the past 30 days using the following API call:

```
      curl -O /tmp/events.json http://localhost:5000/events
```

This call returns a (JSON-encoded) list of user events that we can analyze to calculate our user statistics.

Using this API, we can break our workflow down into two separate tasks: one for fetching user events and another task for calculating the statistics. The data itself can be downloaded using the BashOperator, in a similar fashion as we saw in the previous chapter. For calculating the statistics, we can use a Python operator, which allows us to load the data into a Pandas dataframe and calculate the number of events using a groupby and an aggregation:

```
def calculate_stats(input_path, output_path):
      """Calculates event statistics."""
      events = pd.read_json(input_path)
      stats = events.groupby(["date", "user"]).size().reset_index()
      stats.to_csv(output_path, index=False)
```

Altogether, this gives us the following DAG for our workflow:

```
from airflow import DAG
from airflow.operators.bash_operator import BashOperator
from airflow.operators.python_operator import PythonOperator
from datetime import datetime, timedelta

import pandas as pd

dag = DAG(
     dag_id="user_events",
     start_date=datetime(2015, 6, 1),
     schedule_interval=None
)

fetch_events = BashOperator(
      task_id="fetch_events",
      bash_command="curl -o data/events.json https://localhost:5000/events",
      dag=dag,
)

def _calculate_stats(input_path, output_path):
      """Calculates event statistics."""
      events = pd.read_json(input_path)
      stats = events.groupby(["date", "user"]).size().reset_index()
      stats.to_csv(output_path, index=False)


calculate_stats = PythonOperator(
      task_id="calculate_stats",
      python_callable=_calculate_stats,
      op_kwargs={
    "input_path": "data/events.json",
    "output_path": "data/stats.csv"
},
      dag=dag,
)

fetch_events >> calculate_stats
```

When viewed from Airflow, this should result in a DAG containing two tasks and look something like Figure 3.1.

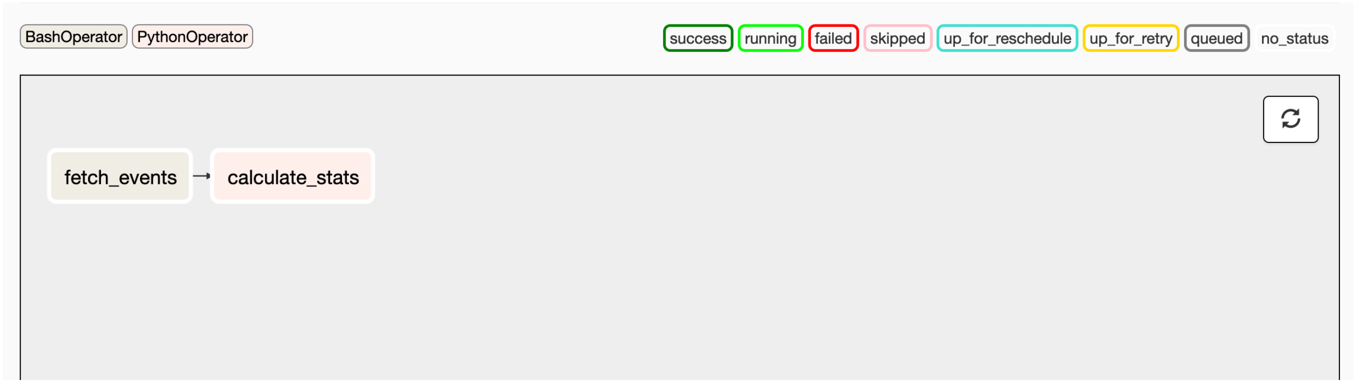


Figure 3.1. Basic DAG for our event use case. In this use case, we first load events from an external API using a bash command (fetch_events) and subsequently summarize these events in several statistics in Python (calculate_stats). In principle, the idea is to run this DAG every day to accumulate a dataset of daily statistics over time.

Now we have our basic DAG, but we still need to make sure it's run regularly by Airflow. Let's get it scheduled so that we have daily updates!

3.1.2 Defining scheduling intervals

In our example of ingesting user events, we would like to calculate statistics on a daily basis, suggesting that it would make sense to schedule our DAG to run once every day. As this is a common use case, Airflow provides the convenient macro '@daily' for defining a daily schedule interval which runs our DAG once every day at midnight:

```
dag = DAG(
      dag_id="user_events",
      schedule_interval="@daily",
      ...
)
```

However, we're not quite done yet. For Airflow to know from which date it should start scheduling our DAG runs, we also need to provide a start date for our DAG. Based on this start date, Airflow will schedule the first execution of our DAG to run at the first schedule interval after the start date (start + interval). Subsequent runs will continue executing at schedule intervals following this first interval.

For example, say we define our DAG with a start date on the first of January:

```
import datetime as dt

dag = DAG(
      dag_id="user_events",
      schedule_interval="@daily",
      start_date=dt.datetime(year=2019, month=1, day=1)
)
```

Combined with a daily scheduling interval, this will result in Airflow running our DAG at midnight every day following the first of January (Figure 3.2). Note that our first execution takes place on the second of January (the first interval following the start date) and not the first of January. We'll dive further into the reasoning behind this behaviour later in this chapter.

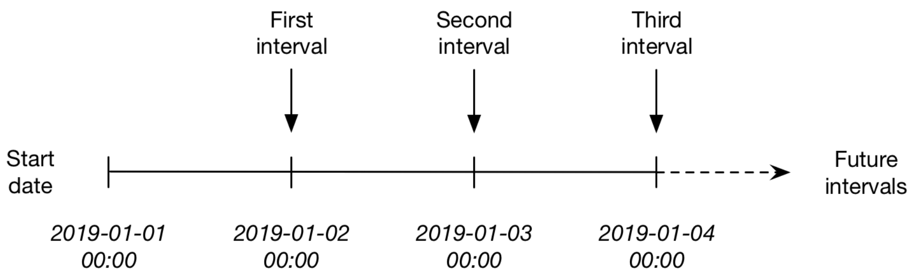


Figure 3.2. Schedule intervals for a daily scheduled DAG with a specified start date. This shows daily intervals for a DAG with a start date of 2019-01-01. Arrows indicate the time point at which a DAG is executed. Without a specified end date, the DAG will keep being executed every day until the DAG is switched off.

Without an end date, Airflow will (in principle) keep executing our DAG on this daily schedule until the end of time. However, if we already know that our project has a fixed duration, we can tell Airflow to stop running our DAG after a certain date using the `end_date` parameter:

```
dag = DAG(
      dag_id="user_events",
      schedule_interval="@daily",
      start_date=dt.datetime(year=2019, month=1, day=1),
     end_date=dt.datetime(year=2019, month=1, day=5)
)
```

This will result in the full set of schedule intervals shown in Figure 3.3.

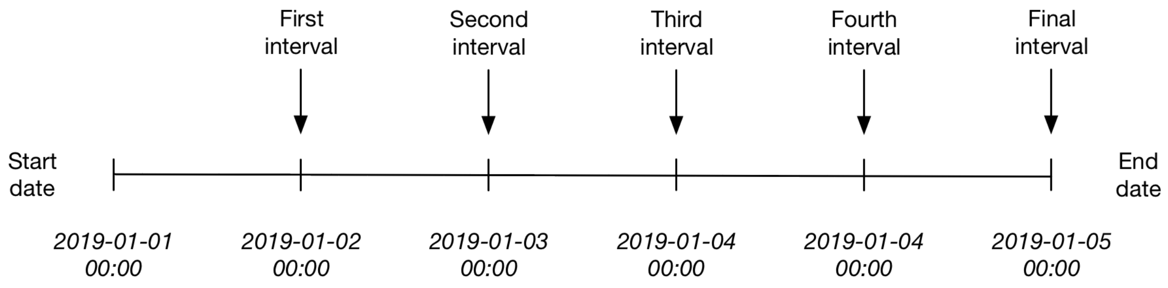


Figure 3.3. Schedule intervals for a daily scheduled DAG with specified start and end dates. Shows intervals for the same DAG as Figure 3.2, but now with an end date of 2019-01-05, which prevents the DAG from executing beyond this date.

In principle, it's good practice to ensure that your start/end dates align with your schedule interval. For example, with a daily schedule interval, it makes more sense to use a start date of 2019-01-01 00:00:00 (midnight) than 2019-01-01 05:50:00, as the former is nicely aligned with the daily schedule interval (which schedules runs at midnight). However, if you use a non-aligned start date Airflow will auto-align auto align the start date and the schedule interval, by using the start date as the moment to start looking for the next schedule interval. In practice, this means that both cases (aligned and non-aligned) will result in Airflow executing its first interval at 2019-01-02 00:00:00.

Besides static dates, it is possible to use dynamic values for start/end dates in Airflow. This is, for example, frequently used in the Airflow tutorial to run DAGs for the past X days, by setting the start_date to something like `airflow.utils.dates.days_ago(14)`. Although this feature is useful for testing or examples, in practice we would recommend against using dynamic start/end dates for your DAGs as they can be confusing. Moreover, certain combinations can result in your DAG never being triggered if they result in Airflow never completing a schedule interval. For example, combining a start date of `datetime.now()` with an hourly schedule interval will result in the DAG never being triggered, as the DAG never reaches an hour after the start date, which moves along with `now`.

3.1.3 Cron-based intervals

Up to now, all our examples have shown DAGs running at daily intervals. But what if we want to run our jobs on hourly or weekly intervals? And what about more complicated intervals in which we, for example, may want to run our DAG at 23:45 every Saturday?

To support more complicated scheduling intervals, Airflow allows us to define scheduling intervals using the same syntax as used by cron, a time-based job scheduler used by Unix-like computer operating systems such as macOS and Linux. This syntax consists of 5 components and is defined as follows:

```
# ┌─────── minute (0 - 59)
# │ ┌────── hour (0 - 23)
# │ │ ┌───── day of the month (1 - 31)
# │ │ │ ┌───── month (1 - 12)
# │ │ │ │ ┌──── day of the week (0 - 6) (Sunday to Saturday;
# │ │ │ │ │      7 is also Sunday on some systems)
# * * * * *
```

In this definition, a cron job is executed when the time/date specification fields match the current system time/date. Asterisks ('*') can be used instead of numbers to define unrestricted fields, meaning that we don't care about the value of that field.

Although this cron-based representation may seem a bit convoluted, it provides us with considerable flexibility for defining time intervals. For example we can define hourly, daily and weekly intervals using the following cron expressions:

- 0 * * * * = hourly (running on the hour)
- 0 0 * * * = daily (running at midnight)
- 0 0 * * 0 = weekly (running at midnight on Sunday)

Besides this, we can also define more complicated expressions such as the following:

- 0 0 1 * * = midnight on the first of every month
- 45 23 * * SAT = 23:45 every Saturday

Additionally, cron expressions allow you to define collections of values using a comma (',') to define a list of values or a dash ('-') to define a range of values. Using this syntax, we can build expressions that enable running jobs on multiple weekdays or multiple sets of hours during a day:

- 0 0 * * MON,WED,FRI = run every Monday, Wednesday, Friday at midnight
- 0 0 * * MON-FRI = run every weekday at midnight
- 0 0,12 * * * = run every day at 00:00 AM and 12:00 P.M.

Airflow also provides support for several macros that represent shorthands for commonly used scheduling intervals. We have already seen one of these macros (@daily) for defining daily intervals. An overview of the other macros supported by Airflow is shown in Table 3.1.

Table 3.1. Airflow presets for frequently used scheduling intervals.

Preset	Meaning	Equivalent cron expression
None (not a string)	Don't schedule the DAG. Used for externally triggered DAGs.	-
@once	Schedule once and only once.	-
@hourly	Run once an hour at the beginning of the hour.	0 * * * *
@daily	Run once a day at midnight.	0 0 * * *
@weekly	Run once a week at midnight on Sunday morning.	0 0 * * 0
@monthly	Run once a month at midnight on the first day of the month.	0 0 1 * *
@yearly	Run once a year at midnight on January 1.	0 0 1 1 *

Although cron expressions are extremely powerful, they can be difficult to work with. As such, it may be a good idea to test your expression before trying it out in Airflow. Fortunately, there are many tools (e.g. https://crontab.guru) available online that can help you define, verify or explain your cron expressions in plain English. It also doesn't hurt to document the reasoning behind complicated cron expressions in your code. This may help others (including future-you!) understand the expression when revisiting your code.

3.1.4 Frequency-based intervals

An important limitation of cron expressions is that they are unable to represent certain frequency-based schedules. For example, how would you define a cron expression that runs a DAG once every three days? It turns out that you could write an expression that runs on every 1st, 4th, 7th, etc. day of the month, but this approach would run into problems at the end of the month as the DAG would run consecutively on both the 31st and the 1st of the next month, violating the desired schedule.

This limitation of cron stems from the nature of cron expressions, as cron expressions define a pattern which is continuously matched against the current time to determine whether a job should be executed or not. This has the advantage of making the expressions stateless, meaning that you don't have to remember when a previous job was run to calculate the next interval. However, as you can see, this comes at the price of some expressiveness.

So what if we really want to run our DAG on a three-daily schedule?

To support this type of frequency-based schedule, Airflow also allows you to define scheduling intervals in terms of a relative time interval. To use such a frequency-based schedule, you can pass a `timedelta` instance (from the datetime module in the standard library) as a schedule interval:

```
from datetime import timedelta

dag = DAG(
    dag_id="user_events",
    schedule_interval=timedelta(days=3),
    start_date=dt.datetime(year=2019, month=1, day=1)
)
```

This would result in our DAG being run every three days following the start date (on the 4th, 7th, 10th etc. of January 2019). Of course, you can also use this approach to run your DAG every 10 minutes (using timedelta(minutes=10)) or every 2 hours (using timedelta(hours=2)).

3.1.5 When to use which interval expression?

Given that Airflow has at least three different ways to define scheduling intervals, you may be wondering when to use which type of expression to define your scheduling interval (preset, cron or frequency-based). In principle, we would recommend using the most simple expression for suitable for your case, as this will produce the most readable result. In practice, this means that we would recommend using Airflow's presets instead of CRON expressions when possible, as these presets are generally considered easier to read. Of course, if you need more granularity than the presets provide, don't hesitate to use a cron expression (but consider adding a short, explanatory comment in your code).

Frequency-based expressions are generally meant to cover intervals not supported by CRON. However, for many cases they may be just as readable as the equivalent preset or CRON expression. For example, using a schedule interval of `timedelta(days=1)` is not much less readable than the using the @daily preset. However, it is important to note that these two schedule intervals are not necessarily the same, depending on the used start date. In the case of a non-rounded start date, such as 2019-01-01 05:00:00, the timedelta based expression will result in the DAG being triggered 1 full day after the start date, at 2019-01-02 05:00:00. In contrast, the @daily preset triggers the DAG to run at midnight, meaning that the DAG would be triggered at 2019-01-02 00:00:00. Depending on your intentions, it's good to keep these small nuances between frequency-based and cron/preset schedules in mind.

3.2 Processing data incrementally

Although we now have our DAG running at a daily interval (assuming we stuck with the @daily schedule), we haven't yet quite achieved our goal. For one, our DAG is downloading and calculating statistics for the entire catalogue of user events every day, which is hardly efficient. Moreover, this process is only downloading events for the past 30 days, which means that we are not building up any history for dates further in the past.

One way to solve these issues is to change our DAG to only fetch and process the data for a single day. This way, we only process the new data that comes in every day and don't waste time re-calculating statistics for previous days, saving our precious computing resources. This type of approach is commonly referred to as incremental processing, as it involves processing small increments of our data for every schedule interval when building up our dataset.

In this section, we will explore how to rewrite our events DAG to load data incrementally. In doing so, we will show you how Airflow scheduling intervals define data increments and the implications incremental processing has on how you store your data.

3.2.1 Fetching events incrementally

To convert our events DAG to use an incremental approach, we need to rewrite our DAG to fetch events and compute statistics on a daily basis, only for the events of the corresponding day. One way to implement this, is to rewrite the fetch task to only fetch events for a single day and write this to an output JSON file containing all the events of that day. Subsequently, the aggregation (calculate_statistics) task can read in the JSON file (events/day1.json) for that day and use the data from that file to calculate statistics for that specific day, which can also be written to an output file containing statistics for that day (stats/day1.csv, Figure 3.4).

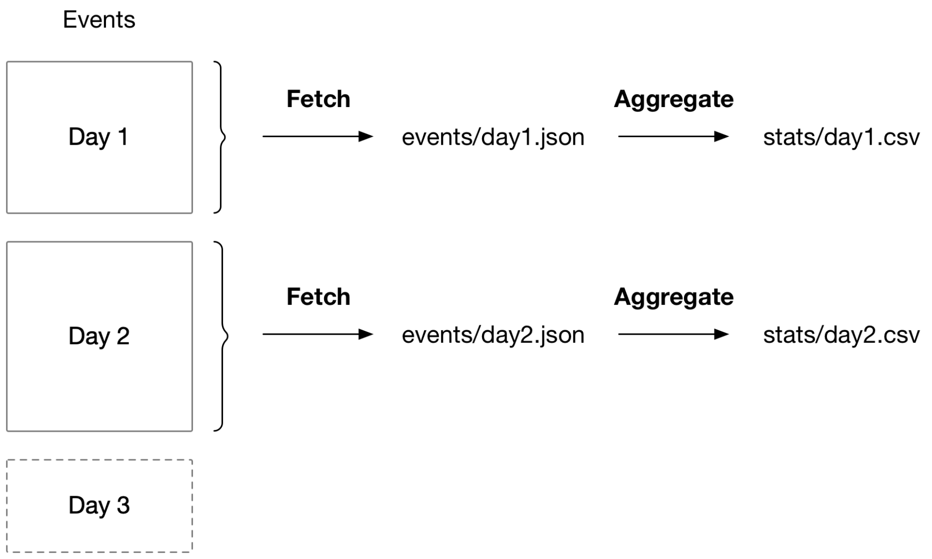


Figure 3.4. Fetching and processing events incrementally. The events DAG can be converted to load and process events incrementally by adjusting the DAG to only fetch and aggregate events for a single day. By repeating this incremental processing of daily data every day, the DAG achieves the same results as before, without having to process the entire dataset. Note that the daily tasks need to write events to individual files for each day, so that we know where to find a days worth of data.

As mentioned before, this incremental approach is much more efficient than fetching and processing the entire dataset, as it significantly reduces the amount of data that has to be processed in each schedule interval. Additionally, because we are now storing our data in separate files per day, we also have the opportunity to start building up a history of files over time, way past the thirty day limit of our API.

To implement incremental processing in our workflow, we need to modify our DAG to download data for a specific day. Fortunately, we can adjust our API call to fetch events for the current date by including start and end date parameters:

```
      curl -O http://localhost:5000/events?start_date=2019-01-01&end_date=2019-01-02
```

Together, these two date parameters indicate the time range for which we would like to fetch events. Note that in this example start_date is inclusive, whilst end_date is exclusive, meaning that we are effectively fetching events that occur between 2019-01-01 00:00:00 and 2019-01-01 23:59:59.

We can implement this incremental data fetching in our DAG by changing our bash command to include the two dates:

```
fetch_events = BashOperator(
       task_id="fetch_events",
       bash_command="curl -o data/events.json
       http://localhost:5000/events?start_date=2019-01-01&end_date=2019-01-02",
       dag=dag,
)
```

However, to fetch data for any other date than 2019-01-01, we need to change the command to use start and end dates that reflect the day for which the DAG is being executed. Fortunately, Airflow provides us with several extra parameters for doing so, which we'll explore in the next section.

3.2.2 Dynamic time references using execution dates

For many workflows involving time-based processes, it is important to know for which time interval a given task is being executed. For this reason, Airflow provides tasks with extra parameters that can be used to determine for which schedule interval a task is being executed.

The most important of these parameters is called the **execution_date**, which represents the date and time for which our DAG is being executed. In contrast to what the name of the parameter suggests, the execution_date is not a date but a timestamp, which reflects the start time of the schedule interval for which the DAG is being executed. The end time of the schedule interval is indicated by another parameter called the next_execution_date. Together these two dates define the entire length of a tasks schedule interval (Figure 3.5).

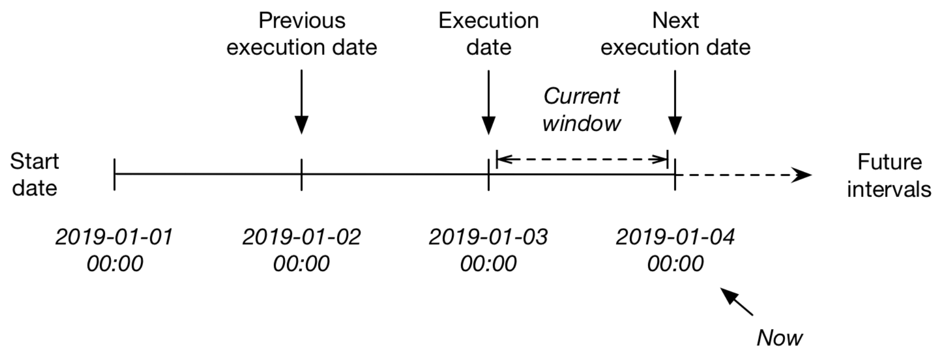


Figure 3.5. Definitions of execution dates in Airflow. Arrows indicate the execution dates defined in the Airflow context and how these relate to the current execution window. Note that the execution date refers to the start of the execution window, whilst the next execution date refers to the next execution date (= current execution date + schedule interval). As such, the current execution window runs between the current and next execution dates.

Besides these two parameters, Airflow also provides a previous_execution_date parameter, which describes the start of the previous schedule interval. Although we won't be using this parameter here, it can be useful for performing analyses that contrast data from the current time interval with results from the previous interval.

In Airflow, we can use these execution dates by referencing them in our operators. For example, in the BashOperator, we can use Airflow's templating functionality to include the execution dates in our dynamically in our bash command:

```
fetch_events = BashOperator(
      task_id="fetch_events",
      bash_command="curl -o data/events.json
      http://localhost:5000/events?start_date={{execution_date.strftime('%Y-%m-
      %d')}}&end_date={{next_execution_date.strftime('%Y-%m-%d')}}",
      dag=dag,
)
```

In this example, the syntax {{variable_name}} is an example of using Airflows jinja-based[4] templating syntax for referencing one of Airflow's specific parameters. Here, we use this syntax to reference both the execution dates and format them to the expected string format using the datetimes strftime method (as both execution dates are datetime objects).

Because the execution date parameters are often used in this fashion to reference dates as formatted strings, Airflow also provides several short hand parameters for common date formats. For example, ds and ds_nodash parameters are different representations of the

[4] http://jinja.pocoo.org/

execution_date, formatted as YYYY-MM-DD and YYYYMMDD respectively. Similarly, the next_ds, next_ds_nodash, prev_ds and prev_ds_nodash provide shorthands for the next and previous execution dates, respectively.

Using these shorthands, we can also write our incremental fetch command as follows:

```
fetch_events = BashOperator(
      task_id="fetch_events",
      bash_command="curl -o data/events.json
      http://localhost:5000/events?start_date={{ds}}&end_date={{next_ds}}",
      dag=dag,
)
```

This shorter version is quite a bit easier to read. However, for more complicated date (or datetime) formats, you will likely still need to use the more flexible strftime approach.

3.2.3 Partitioning your data

Although our new fetch_events task now fetches events incrementally for each new schedule interval, the astute reader may have noticed that each new task is simply overwrite the result of the previous day, meaning that we are effectively not building up any history.

One way to solve this problem is to simply append new events to the events.json file, which would allow us to build up our history in a single JSON file. However, a drawback of this approach is that it requires any downstream processing jobs to load the entire dataset, even if we are only interested in calculating statistics for a given day. Additionally, it also makes this file a single point of failure, by which we may risk losing our entire dataset should this file become lost of corrupted.

An alternative approach is to divide our dataset into daily batches by writing the output of the task to a file bearing the name of the corresponding execution date:

```
fetch_events = BashOperator(
      task_id="fetch_events",
      bash_command="curl -o data/events/{{ds}}.json
      http://localhost:5000/events?start_date={{ds}}&end_date={{next_ds}}",
      dag=dag,
)
```

This would result in any data being downloaded for an execution date of 2019-01-01 being written to the file data/2019-01-01.json.

This practice of dividing a dataset into smaller, more manageable pieces is a common strategy in data storage and processing systems. The practice is commonly referred to as partitioning, with the smaller pieces of a dataset being referred to as partitions.

The advantage of partitioning our dataset by execution date becomes evident when we consider the second task in our DAG (calculate_stats), in which we calculate statistics for each day’s worth of user events. In our previous implementation, we were loading the entire dataset and calculating statistics for our entire event history, every day:

```
def _calculate_stats(input_path, output_path):
      """Calculates event statistics."""
      events = pd.read_json(input_path)
      stats = events.groupby(["date", "user"]).size().reset_index()
      stats.to_csv(output_path, index=False)


calculate_stats = PythonOperator(
      task_id="calculate_stats",
      python_callable=_calculate_stats,
      op_kwargs={
   "input_path": "data/events.json",
   "output_path": "data/stats.csv"
},
      dag=dag,
)
```

However, using our partitioned dataset, we can calculate these statistics more efficiently for each separate partition by changing the input and output paths of this task to point to the partitioned event data and a partitioned output file:

```
def _calculate_stats(**context):
      """Calculates event statistics."""

      input_path = context["templates_dict"]["input_path"]
      output_path = context["templates_dict"]["output_path"]

      events = pd.read_json(input_path)
      stats = events.groupby(["date", "user"]).size().reset_index()
      stats.to_csv(output_path, index=False)


calculate_stats = PythonOperator(
      task_id="calculate_stats",
      python_callable=_calculate_stats,
      templates_dict={
    "input_path": "data/events/{{ds}}.json",
    "output_path": "data/stats/{{ds}}.csv"
},
provide_context=True,
      dag=dag,
)
```

Although these changes may look somewhat complicated, they mostly involve boilerplate code for ensuring that our input and output paths are templated. To achieve this templating in the PythonOperator, we need to pass any arguments that should be templated using the operators templates_dict parameter (lines …). To retrieve these templated arguments in our Python function, we need to retrieve their values from the task context (lines .. and ..) after making sure that the context is passed to our function (line …).

If this all went a bit too quickly, don’t worry - we’ll dive into the task context in more detail in the next chapter. The important point to understand here is that these changes allow us to compute our statistics incrementally, by only processing a small subset of our data each day.

3.3 Understanding Airflow's execution dates

Because execution dates are such an important part of Airflow, let's take a minute to make sure that we fully understand how these dates are defined.

As we've seen, we can control when Airflow runs a DAG with three parameters: a start date, a schedule interval and an (optional) end date. To actually start scheduling our DAG, Airflow uses these three parameters to divide time into a series of schedule intervals, starting from the given start date and optionally ending at the end date (Figure 3.6).

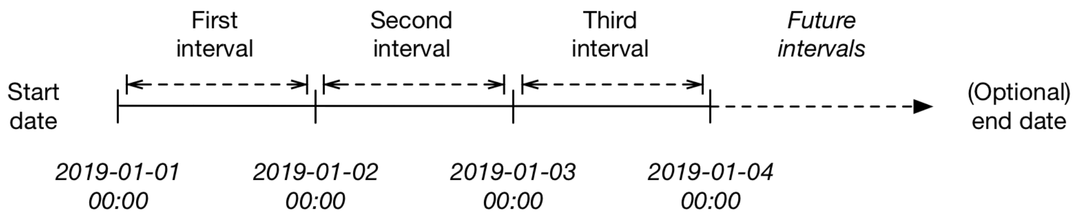


Figure 3.6. Time represented in terms of Airflow's scheduling intervals. Assumes a daily interval with a start date of 2019-01-01. Scheduling intervals run from the DAGs start date up to an optional end date.

In this interval-based representation of time, a DAG is executed for a given interval as soon as the time slot of that interval has passed. For example, the first interval in Figure 3.6 would be executed as soon as possible after 2019-01-01 23:59:59, as by then the last time point in the interval has passed. Similarly, the DAG would execute for the second interval shortly after 2019-01-02 23:59:59 and so on, until we reach our optional end date.

An advantage of using this interval-based approach, is that it is ideal for performing the type of incremental data processing that we saw in the previous sections, as we know exactly for which interval of time a task is executing for - the start and end of the corresponding interval. This is in stark contrast to for example a time point-based scheduling system such as cron, where we only know the current time for which our task is being executed. This means that, for example in cron, we either have to calculate or 'guess' where our previous execution left off, by for example assuming that the task is executing for the previous day (Figure 3.7).

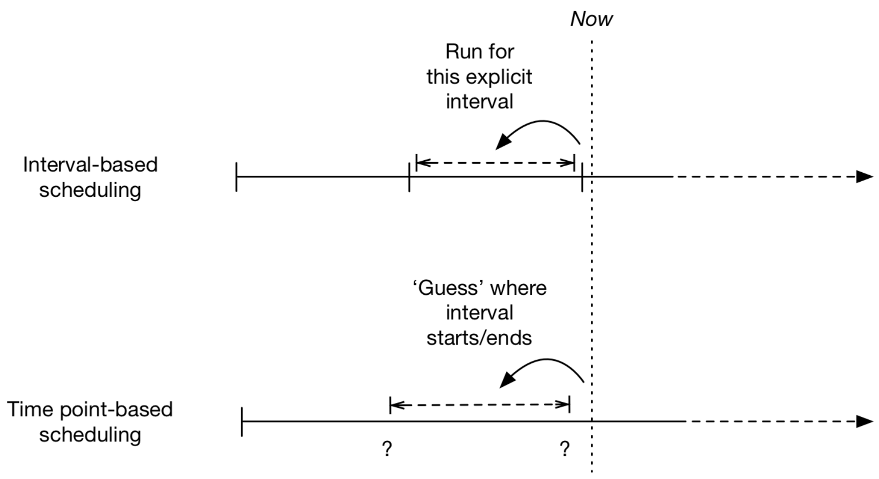

Now
Run for this explicit interval
Interval-based scheduling
'Guess' where interval starts/ends
Time point-based scheduling
?
?

Figure 3.7. Incremental processing in interval-based scheduling windows (e.g. Airflow) vs windows derived from time point based systems (e.g. cron). For incremental (data) processing, time is typically divided into discrete time intervals which are processed as soon as the corresponding interval passed. Interval-based scheduling approaches (such as Airflow) explicitly schedule tasks to run for each interval, whilst providing exact information to each task concerning the start and the end of the interval. In contrast, time point-based scheduling approaches only execute tasks at a given time, leaving it up to the task itself to determine for which incremental interval the task is executing.

Understanding that Airflow's handling of time is built around schedule intervals also helps understand how execution dates are defined within Airflow. For example, say we have a DAG following a daily schedule interval and consider the corresponding interval that should process data for the day 2019-01-03. In Airflow, this interval will be run shortly after 2019-01-04 00:00:00, as at that point in time we know that we will no longer be receiving any new data for the day of 2019-01-03. Thinking back to our explanation of using execution dates in our tasks from the previous section, what do you think that the value of execution_date will be for this interval?

What many people expect is that the execution date of this DAG run will be 2019-01-04, as this is the moment at which the DAG is actually run. However, if we look at the value of the execution_date variable when our tasks are executed, we will actually see an execution date of 2019-01-03. This is because Airflow defines the execution date of a DAG as the start of the corresponding interval. Conceptually, this makes sense if we consider that the execution date marks our schedule interval, rather than the moment on which our DAG is actually executed. Unfortunately, the naming can be a bit confusing.

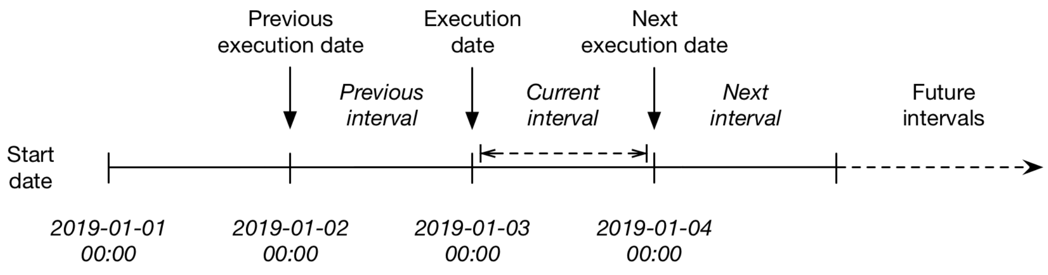


Figure 3.8. Execution dates in the context of schedule intervals. In Airflow, the execution date of a DAG is defined as the start time of the corresponding schedule interval, rather than the time at which the DAG is executed (which is typically the end of the interval). As such, the value of execution_date points to the start of the current interval, whilst the previous_execution_date and next_execution_date parameters point to the start of the previous and next schedule intervals, respectively. The current interval can be derived from a combination of the execution_date and the next_execution_date, which signifies the start of the next interval and thus the end of the current interval.

With Airflow execution dates being defined as the start of the corresponding schedule intervals, they can be used to derive the start and end of a specific interval (Figure 3.8). For example, when executing a task, the start and end of the corresponding interval are defined by the execution_date (the start of the interval) and the next_execution date (the start of the next interval) parameters. Similarly, the previous schedule interval can be derived using the previous_execution_date and execution_date parameters.

However, one caveat to keep in mind when using the previous_execution_date and next_execution_date parameters in your tasks is that these parameters are only defined for DAG runs following the schedule interval. As such, the values of these parameters will be undefined for any runs that are triggered manually using Airflow UI or CLI. The reason for this is that Airflow cannot provide you with information about next or previous schedule intervals if you are not following a schedule interval.

3.4 Using backfilling to fill in past gaps

As Airflow allows us to define schedule intervals starting from an arbitrary start date, we can also define past intervals starting from a start date in the past. We can use this property to perform historical runs of our DAG for loading or analyzing past datasets - a process typically referred to as backfilling (Figure 3.9).

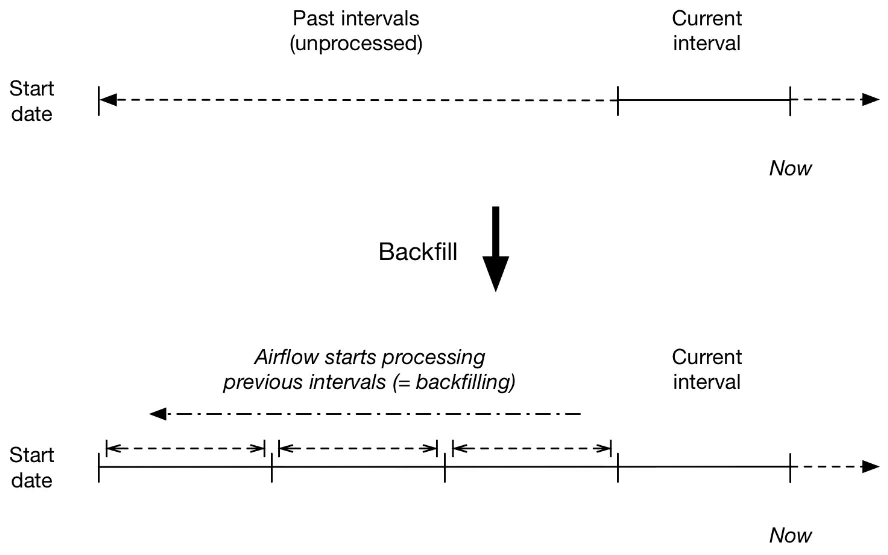


Figure 3.9. Using backfilling to fill up past gaps. Because of the way that Airflow defines schedule intervals, it also has the ability to execute runs for previous schedule intervals in the past. Executing runs for schedule intervals is typically referred to as backfilling, as backfilling allows you to fill in (past) holes in your dataset.

By default, Airflow will schedule and run any past schedule intervals that have not yet been run. As such, specifying a past start date and activating the corresponding DAG will result in all intervals that have passed before the current time being executed. This behaviour is controlled by the DAG *catchup* parameter and can be disabled by setting catchup to False:

```
dag = DAG(
    dag_id="user_events",
    schedule_interval=timedelta(days=3),
    start_date=dt.datetime(year=2019, month=1, day=1),
    catchup=False
)
```

With this setting, the DAG will only be run for the most recent schedule interval, rather than executing all open past intervals. The default value for *catchup* can be controlled from the Airflow configuration file, by setting a value for the *catchup_by_default* configuration setting.

Although backfilling is a powerful concept, it is limited by the availability of data in source systems. For example, in our example use case we can load past events from our API by specifying a start date up to 30 days in the past. However, as the API only provides up to 30 days of history, we cannot use backfilling to load data from earlier days.

Backfilling can also be used to re-process data after we have made changes in our code. For example, say we make a change to our *_calc_statistics* function to add a new statistic. Using backfilling, we can clear past runs of our *calc_statistics* task to re-analyze our historical data using the new code. Note that in this case we aren't limited by the 30 day limit of our data source, as we have already loaded these earlier data partitions as part of our past runs.

3.5 Best practices for designing tasks

Although Airflow does much of the heavy lifting when it comes to backfilling and re-running tasks, we need to make sure that our tasks fulfill certain key properties to ensure proper results. In this section, we will dive into two of the most important properties of proper Airflow tasks: atomicity and idempotency.

3.5.1 Atomicity

The term atomicity is frequently used in database systems, where an atomic transaction is considered to be an indivisible and irreducible series of database operations such that either all occur, or nothing occurs. Similarly, in Airflow, tasks should be defined to that they either succeed and produce some proper end result, or fail in a manner that does not affect the state of the system.

As an example, consider a simple extension to our user event DAG, in which we would like to add some functionality that sends an e-mail of our top 10 users at the end of each run. One simple way to add this would be to extend our previous function with an additional call to some function that sends an email containing our statistics:

```
def _calculate_stats(**context):
      """Calculates event statistics."""

      input_path = context["templates_dict"]["input_path"]
      output_path = context["templates_dict"]["output_path"]

      events = pd.read_json(input_path)
      stats = events.groupby(["date", "user"]).size().reset_index()
      stats.to_csv(output_path, index=False)

      email_stats(stats, email="user@example.com")
```

Unfortunately, a drawback of this approach is that the task is no longer atomic. Can you see why? If not, consider what happens if our send_stats function fails (which is bound to happen if our email server is a bit flaky). In this case, we will already have written our statistics to the output file at *output_path*, making it seem as if our task succeeded even though it ended in failure (Figure 3.10A).

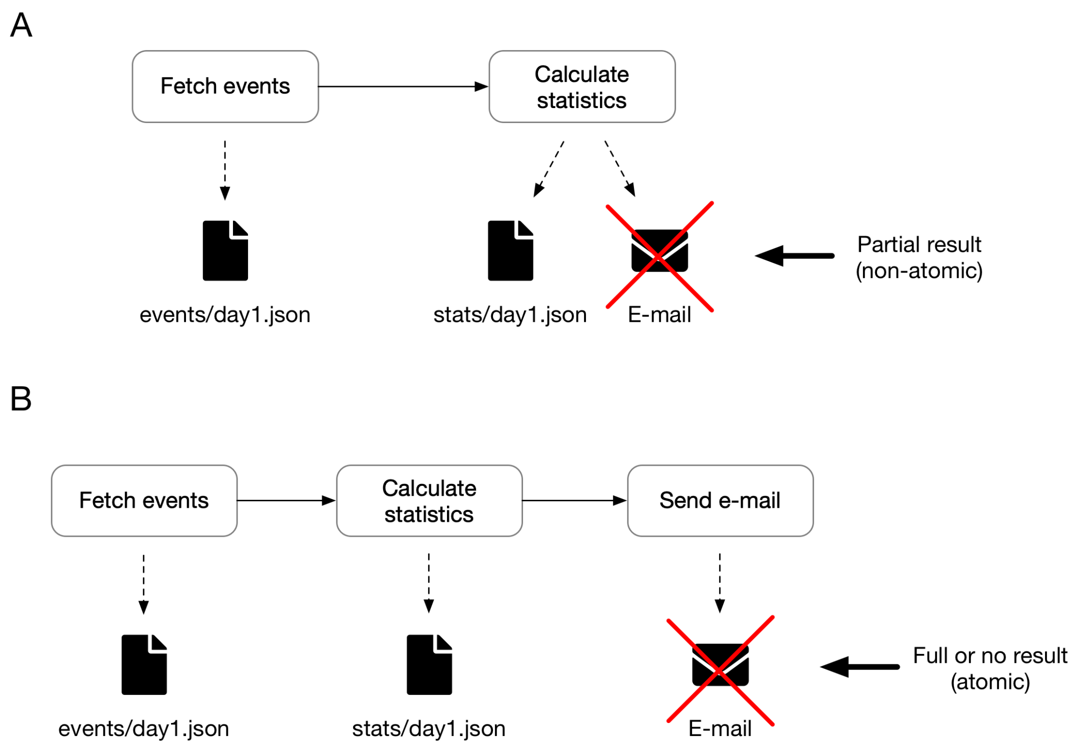


Figure 3.10. Illustration of an atomic and non-atomic implementation for our events DAG. (A) Adding an e-mail notification to the 'calculate statistics' task may result in situations where the task succeeds in writing the output file but fails in sending the e-mail, in which case the overall task has failed. However, because the output file was already written before the failure, we are left with a partial output from the task. As such, this task is not atomic, as atomic tasks should either succeed or fail without any side effect. **(B)** By adding the email notification as a separate task, we can make both tasks are atomic. In this case, if the notification task fails, the entire task fails without any effect on our system and can thus be considered atomic. Similarly, the 'calculate statistics' task also either succeeds in writing the output file or fails and is therefore atomic.

To implement this functionality in an atomic fashion, we could simply split the email functionality out into a separate task:

```
def _send_stats(email, **context):
       stats = pd.read_csv(context["templates_dict"]["stats_path"])
       email_stats(stats, email=email)


send_stats = PythonOperator(
       task_id="send_stats",
       python_callable=_send_stats,
       op_kwargs={
           "email": "user@example.com"
       },
       template_dict={
           "stats_path": "data/stats/{{ds}}.csv"
       },
       provide_context=True,
       dag=dag,
)

calculate_stats >> send_stats
```

This way, failing to send an email no longer affects the result of the *calculate_stats* task, but only fails *send_stats*, thus making both tasks atomic (Figure 3.10B).

From this example, you might think that separating all operations into individual tasks is sufficient to make all our tasks atomic. This is however not necessarily true. To see why, think about what would if our event API would require us to login before querying for events. This would generally require an extra API call to fetch some authentication token, after which we can start retrieving our events.

Following our previous reasoning of one operation = one task, we would have to split these operations into two separate tasks. However, doing so would create a strong dependency between the two tasks, as the second task (fetching the events) will fail without running the first shortly before. This strong dependency between the two tasks means that we are likely better off keeping both operations within a single task, allowing the task to form a single coherent unit of work.

Most Airflow operators are already designed to be atomic, which is why many operators include options for performing tightly coupled operations such as authentication internally. However, more flexible operators such as the Python and Bash operators may require you to think carefully about your operations to make sure that your tasks remain atomic.

3.5.2 Idempotency

Besides atomicity, another important property to consider when writing Airflow tasks is idempotency. Tasks are said to be idempotent if calling the same task multiple times with the same inputs has no additional effect. This means, for example, that re-running a task without changing the inputs should not change the overall output.

For example, consider our last implementation of the *fetch_events* task, which fetches the results for a single day and writes this to our partitioned dataset:

```
fetch_events = BashOperator(
      task_id="fetch_events",
      bash_command="curl -o data/events/{{ds}}.json
      http://localhost:5000/events?start_date={{ds}}&end_date={{next_ds}}",
      dag=dag,
)
```

Re-running this task for a given date would result in the task fetching the same set of events as its previous execution (assuming the date is within our 30 day window) and overwrite the existing JSON file in the data/events folder, producing the same end result. As such, this implementation of the fetch events task is clearly idempotent (Figure 3.11A).

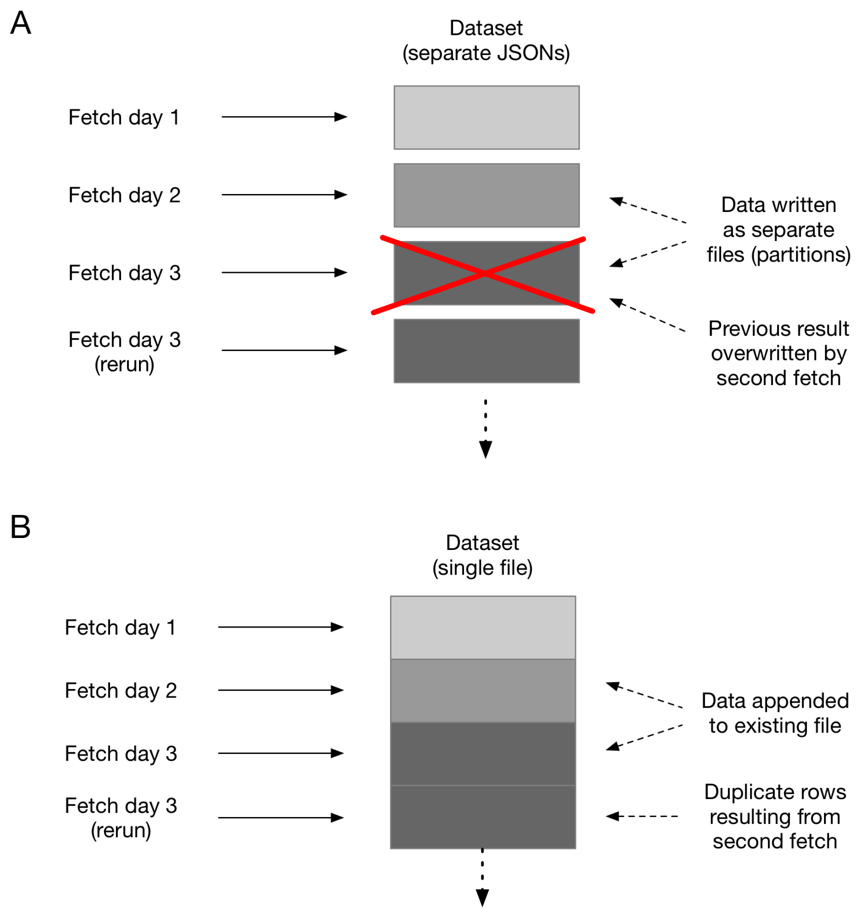


Figure 3.11. Idempotency in our events DAG. (A) When writing daily results to individual files, rerunning the fetch task for a given day results in the day's JSON file being overwritten. This overwrite ensures that we have the same result as before running the task: three JSON files containing data from days 1 - 3. **(B)** Conversely, if we were to use a single dataset to store our results, our implementation would likely involve appending data to this file whenever we fetch data for a given day. In this case, a naive implementation would result in duplicate rows if we were to re-run one of the fetch tasks, as we end up appending the same data to the file twice. As such, re-running the task affects the overall state of the system, thus making the task non-idempotent.

For an example of a non-idempotent task, consider the situation in which we discussed using a single JSON file (data/events.json) and simply appending events to this file. In this case, re-running a task would result in the events simply being appended to the existing dataset, thus duplicating the days events in the dataset. As such, this implementation is not idempotent, as additional executions of the task change the overall result (Figure 3.11B).

In general, tasks that write data can be made idempotent by checking for existing results or making sure that previous results are overwritten by the task. In time-partitioned datasets this is relatively straightforward, as we can simply overwrite the corresponding partition. Similarly, for database systems we can use upsert operations to insert data, which allows us to overwrite existing rows that we written by previous task executions. However, in more general applications you should carefully consider all side effects of your task and make sure that all these side effects are performed in an idempotent fashion.

3.6 Summary

- Schedule a DAG in Airflow using the start_date, schedule_interval and end_date properties of a DAG.
- Cron- or frequency-based expressions to schedule your DAG at the appropriate intervals.
- Use the time windows defined by Airflow's schedule intervals to perform incremental processing of your dataset, which allows you to efficiently process subsets of your data as they come in.
- Use the different execution date parameters provided by Airflow to reference specific subsets of your (partitioned) dataset.
- Execution dates are defined in terms of schedule intervals, meaning that the execution date of a schedule interval refers to the start of the interval, rather than the moment that the DAG is actually executed.
- Use backfilling to fill in past gaps in your dataset.
- Properly designed Airflow tasks should be atomic and idempotent to ensure that results remain correct when re-running tasks.