In the previous chapter, we learned why working with data and the many tools in the data landscape is not easy. In this chapter, we get started with Airflow and check out an example workflow that uses basic building blocks found in many workflows.

2.1 Collecting data from numerous sources

Rockets are one of humanity's engineering marvels, and every rocket launch attracts attention all around the world. In this tutorial, 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 2 (https://thespacedevs.com/llapi), an online repository of data about both historical and future rocket launches from various sources. It is a free and open API for anybody on the planet (subject to rate limits).

John is currently only interested in upcoming rocket launches. Luckily, the Launch Library provides exactly the data he is looking for (https://ll.thespacedevs.com/2.0.0/launch/upcoming). It provides data about upcoming rocket launches, together with URLs of where to find images of the respective rockets. Here's a snippet of the data this URL returns.

```
$ curl -L "https://ll.thespacedevs.com/2.0.0/launch/upcoming"

{
 ...
 "results": [
   {
     "id": "528b72ff-e47e-46a3-b7ad-23b2ffcec2f2",
     "url": "https://.../528b72ff-e47e-46a3-b7ad-23b2ffcec2f2/",
     "launch_library_id": 2103,
     "name": "Falcon 9 Block 5 | NROL-108",
     "net": "2020-12-19T14:00:00Z",
     "window_end": "2020-12-19T17:00:00Z",
     "window_start": "2020-12-19T14:00:00Z",
     ➥ "image":
     "https://spacelaunchnow-prod-east.nyc3.digitaloceanspaces.com/
      media/launch_images/falcon2520925_image_20201217060406.jpeg",
     "infographic": ".../falcon2520925_infographic_20201217162942.png",
     ...
   },
   {
     "id": "57c418cc-97ae-4d8e-b806-bb0e0345217f",
     "url": "https://.../57c418cc-97ae-4d8e-b806-bb0e0345217f/",
     "launch_library_id": null,
     "name": "Long March 8  | XJY-7 & others",
     "net": "2020-12-22T04:29:00Z",
     "window_end": "2020-12-22T05:03:00Z",
     "window_start": "2020-12-22T04:29:00Z",
     "image": "https://.../long2520march_image_20201216110501.jpeg",
     "infographic": null,
     ...
   },
   ...
 ]
}
```

1 2 Inspect the URL response with curl from the command line.

2 3 4 The response is a JSON document, as you can see by the structure.

3 4 The square brackets indicate a list.

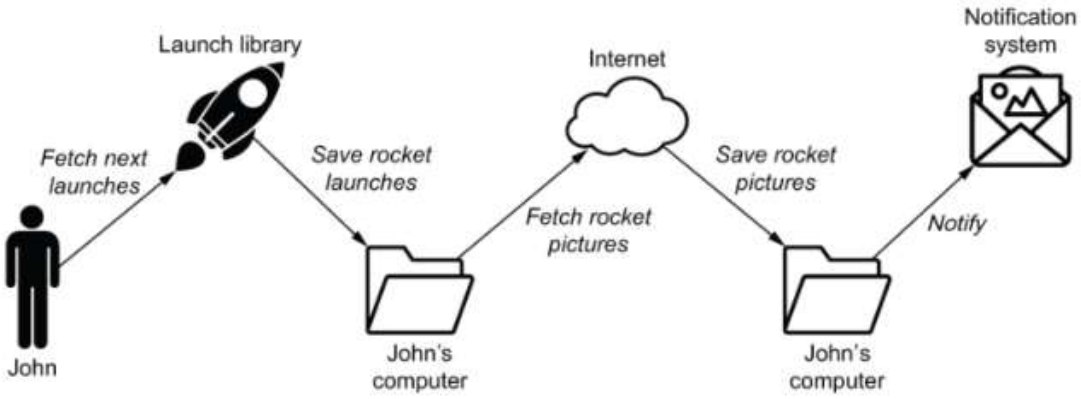
4 All values within these curly braces refer to one single rocket launch.

5 Here we see information such as rocket ID and start and end time of the rocket launch window.

```
6 A URL to an image of the launch-
  ing rocket
```

As you can see, the data is in JSON format and provides rocket launch information, and for every launch, there's information about the specific rocket, such as ID, name, and the image URL. This is exactly what John needs, and he initially draws the plan in figure 2.1 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



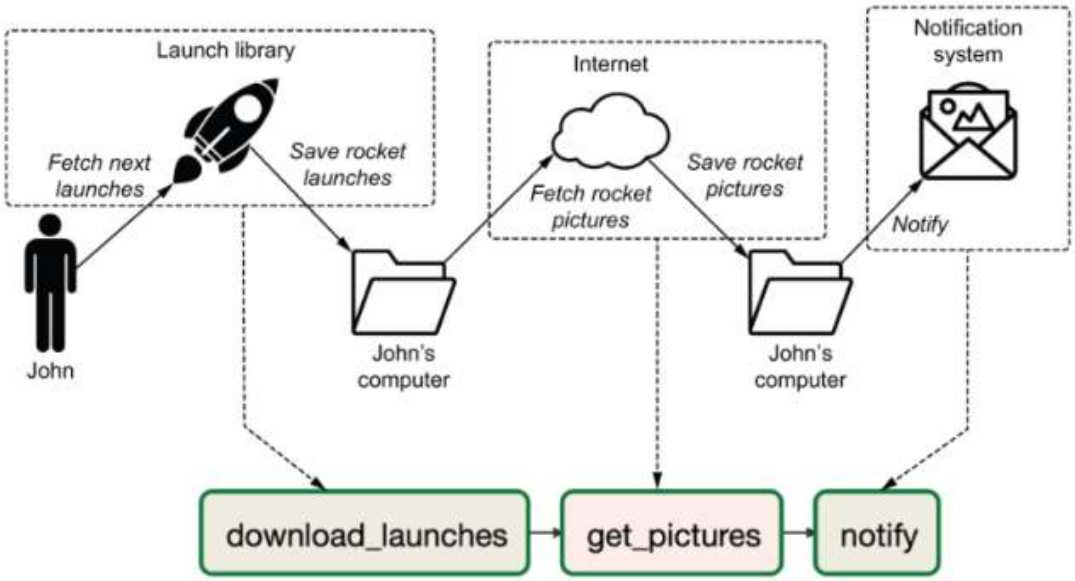
Based on the example in figure 2.1, we can see that, at the end of the day, John's goal is to have a directory filled with rocket images.

2.2 Writing 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-coding, 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" that together form a 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 in figure 2.3.

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



The code for this workflow is as follows.

```
import json
import pathlib

import airflow
import requests
import requests.exceptions as requests_exceptions
from airflow import DAG
from airflow.operators.bash import BashOperator
from airflow.operators.python import PythonOperator

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

download_launches = BashOperator(
    task_id="download_launches",
    bash_command="curl -o /tmp/launches.json -L 'https://ll.thespacedevs.com/2.0.0/launc
    dag=dag,
)
```

```
22
23
24 def _get_pictures():                                                    7
25     # Ensure directory exists
26     pathlib.Path("/tmp/images").mkdir(parents=True, exist_ok=True)
27
28     # Download all pictures in launches.json
29     with open("/tmp/launches.json") as f:
30         launches = json.load(f)
31         image_urls = [launch["image"] for launch in launches["results"]]
32         for image_url in image_urls:
33             try:
34                 response = requests.get(image_url)
35                 image_filename = image_url.split("/")[-1]
36                 target_file = f"/tmp/images/{image_filename}"
37                 with open(target_file, "wb") as f:
38                     f.write(response.content)
39                 print(f"Downloaded {image_url} to {target_file}")
40             except requests_exceptions.MissingSchema:
41                 print(f"{image_url} appears to be an invalid URL.")
42             except requests_exceptions.ConnectionError:
43                 print(f"Could not connect to {image_url}.")
44
```

```
45
46 get_pictures = PythonOperator(                                          8
47     task_id="get_pictures",
48     python_callable=_get_pictures,
49     dag=dag,
50 )
51
52 notify = BashOperator(
53     task_id="notify",
54     bash_command='echo "There are now $(ls /tmp/images/ | wc -l) images."',
55     dag=dag,
56 )
57
58 download_launches >> get_pictures >> notify                             9
```

1 2 3 4 Instantiate a DAG object; this is the starting point of any workflow.

2 3 The name of the DAG

3 4 The date at which the DAG should first start running

4 At what interval the DAG should run

5 6 Apply Bash to download the URL response with curl.

6 The name of the task

7 A Python function will parse the response and download all rocket pictures.

8 Call the Python function in the DAG with a PythonOperator.

9 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 Instantiating a DAG object

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

1 2 3 The DAG class takes two required arguments.

2 3 The name of the DAG displayed in the Airflow user interface (UI)

3 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 `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.

Also note we set `schedule_interval` to `None`. This means the DAG will not run automatically. For now, you can trigger it manually from the Airflow UI. We will get to scheduling later.

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 Instantiating a `BashOperator` to run a Bash command

```
download_launches = BashOperator(
    task_id="download_launches",
    bash_command="curl -o /tmp/launches.json 'https://
ll.thespacedevs.com/2.0.0/launch/upcoming'",
    dag=dag,
)
```

1 The name of the task

2 The Bash command to execute

3 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 independently 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. To make sure the tasks run in the correct order, we can set dependencies between tasks.

Listing 2.5 Defining the order of task execution

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

In Airflow, we can use the *binary right shift operator* (i.e., *"rshift"* [>>]) to define dependencies between tasks. This ensures the `get_pictures` task runs only after `download _launches` has completed successfully, and the `notify` task runs only after `get_pictures` has completed successfully.

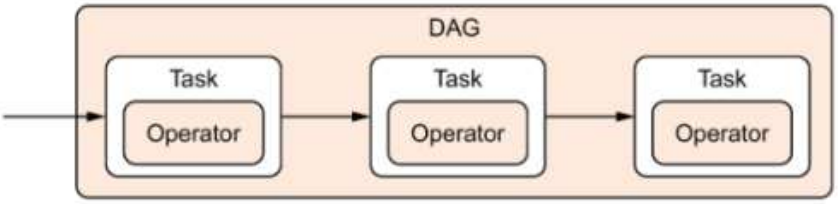
2.2.1 Tasks vs. operators

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, and so on.

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 `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).

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 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 require a bit more effort. Although all this is still possible in a Bash one-liner, it's 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's convenient to keep both the workflow and execution logic in the same script. For downloading the rocket pictures, we implemented listing 2.6.

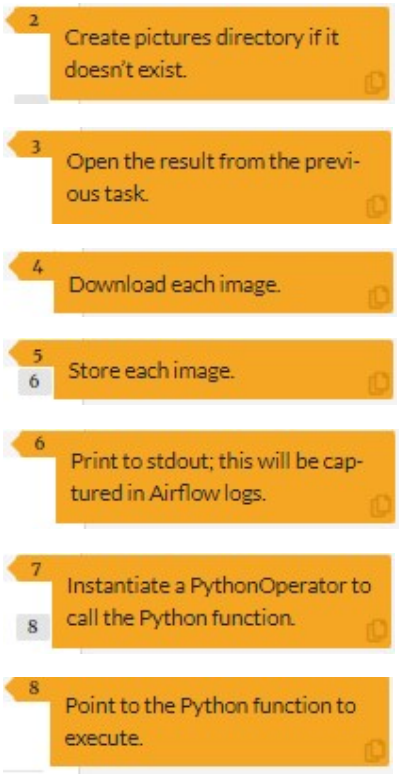
Listing 2.6 Running a Python function using the `PythonOperator`

```
def _get_pictures():                                                    1
    # Ensure directory exists
    pathlib.Path("/tmp/images").mkdir(parents=True, exist_ok=True)      2

    # Download all pictures in launches.json
    with open("/tmp/launches.json") as f:                               3
        launches = json.load(f)
        image_urls = [launch["image"] for launch in launches["results"]]
        for image_url in image_urls:
            try:
                response = requests.get(image_url)                      4
                image_filename = image_url.split("/")[-1]
                target_file = f"/tmp/images/{image_filename}"
                with open(target_file, "wb") as f:
                    f.write(response.content)                           5
                print(f"Downloaded {image_url} to {target_file}")       6
            except requests_exceptions.MissingSchema:
                print(f"{image_url} appears to be an invalid URL.")
            except requests_exceptions.ConnectionError:
                print(f"Could not connect to {image_url}.")


get_pictures = PythonOperator(                                          7
    task_id="get_pictures",
    python_callable=_get_pictures,                                      8
    dag=dag,
)
```

1 Python function to call

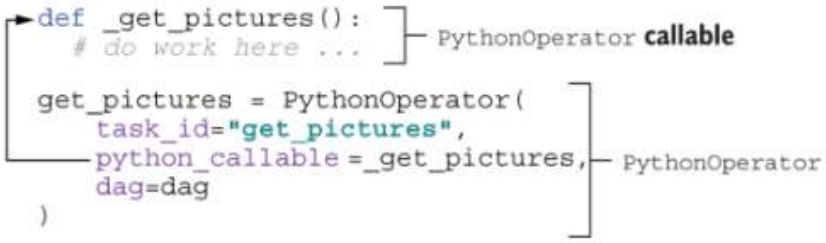


The `PythonOperator` in Airflow is responsible for running any Python code. Just like the `BashOperator` used before, this and all other operators require a `task_id`. The `task_id` is referenced when running a task and displayed in the UI. The use of a `PythonOperator` is always twofold:

1. We define the operator itself (`get_pictures`).
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 like figure 2.5.

Figure 2.5 The `python_callable` argument in the `PythonOperator` points to a function to execute.



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

Listing 2.7 Ensures that the output directory exists and creates it if it doesn't

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

The 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.

Listing 2.8 Extracts image URLs for every rocket launch

```
with open("/tmp/launches.json") as f:                                    1
    launches = json.load(f)                                              2
    image_urls = [launch["image"] for launch in launches["results"]]     3
```

1 2 Open the rocket launches' JSON.

2 3 Read as a dict so we can mingle the data.

3 For every launch, fetch the element "image".

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

Listing 2.9 Downloads all images from the retrieved image URLs

```
for image_url in image_urls:
    try:
        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}")
    except requests_exceptions.MissingSchema:
        print(f"{image_url} appears to be an invalid URL.")
    except requests_exceptions.ConnectionError:
        print(f"Could not connect to {image_url}.")
```

1 Loop over all image URLs.

2 3 Get the image.

3 4 5 6 7 8 Get only the filename by selecting everything after the last. For example, https://host/RocketImages/Electron.jpg_1440.jpg ® Electron.jpg_1440.jpg.

4 5 Construct the target file path.

5 6 Open target file handle.

6 7 Write image to file path.

7 Print result.

8 Catch and process potential errors.

2.3 Running a DAG in Airflow

Now that 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 three core components: a scheduler, a webserver, and a database.

start Airflow 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 ~Desktop/airflow-tutorial/dags/`
3. `airflow webserver`
4. `airflow scheduler`

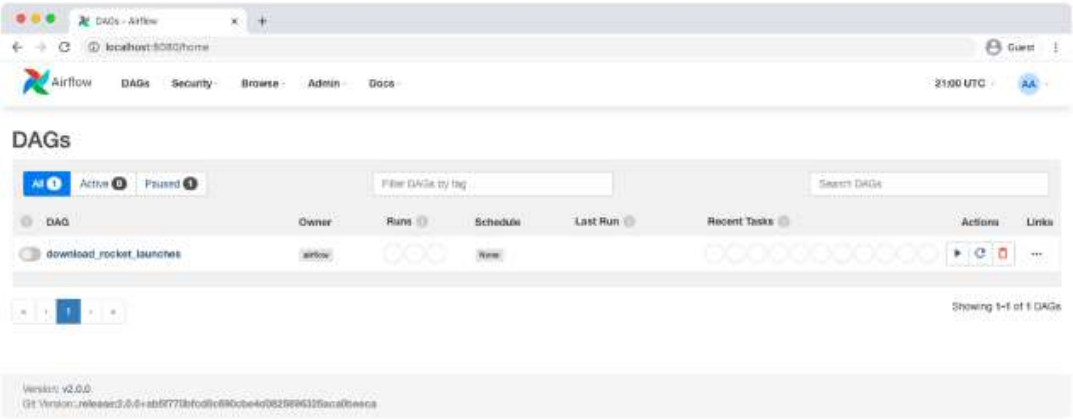
`Note – Make sure to activate the conda virtual environment and then give above commands.`

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

Lab - `download_rocket_launches.py`

After logging in, you can inspect the download_rocket_launches DAG, as shown in figure 2.7.

Figure 2.7 Airflow home screen



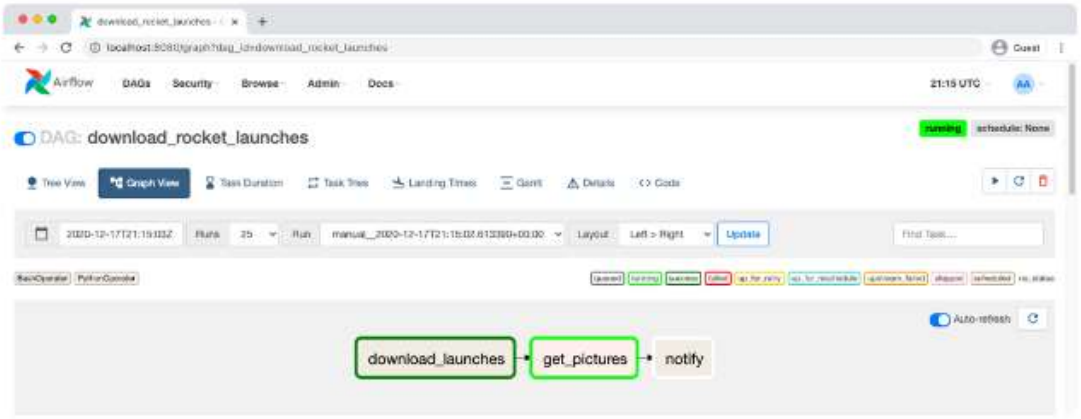
This is the first glimpse of Airflow you will see. Currently, the only DAG is the download_rocket_launches, which is available to Airflow in the DAGs directory. There's a lot of information on the main view, but let's inspect the

download_rocket _launches DAG first. Click on the DAG name to open it and inspect the so-called graph view (figure 2.8).

Figure 2.8 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, and 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.

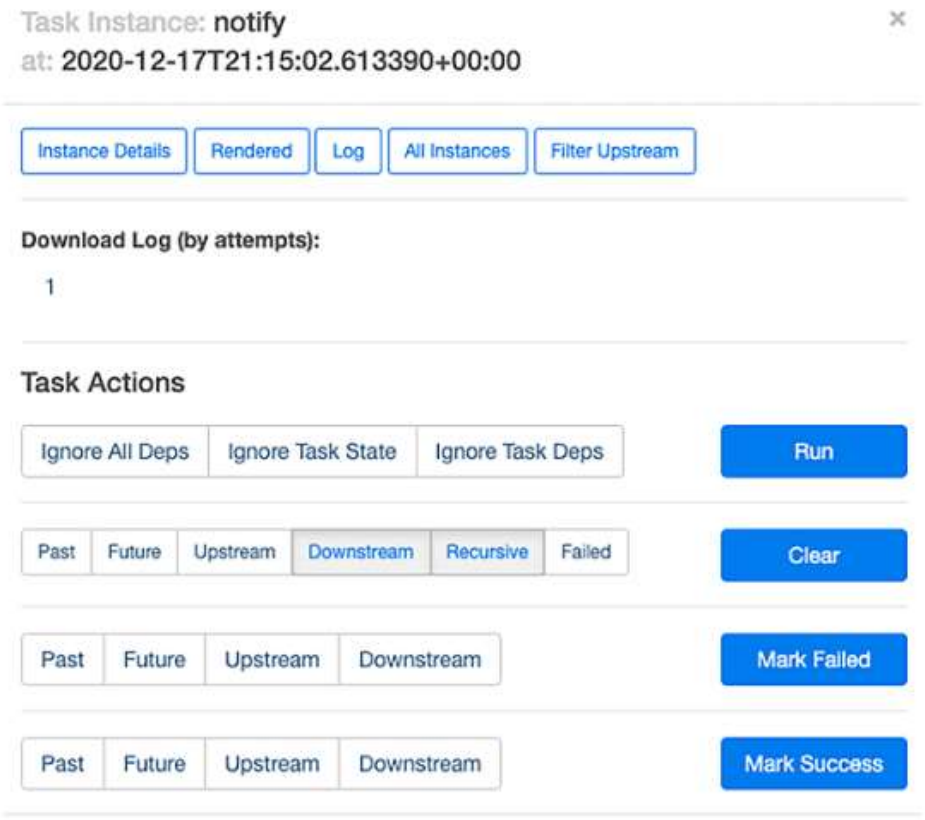
Figure 2.9 Graph view displaying a running DAG



After triggering the DAG, it will start running and you will see the current state of the workflow represented by colors (figure 2.9). 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, for example, 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.

Figure 2.10 Task pop-up options



All task logs are collected in Airflow, so we can search in the UI for output or potential issues in case of failure. Click on a completed notify task, and you will see a pop-up with several options, as shown in figure 2.10.

Click on the top-center Log button to inspect the logs, as shown in figure 2.11.

Figure 2.11 Print statement displayed in logs

```
*** Reading local file: /opt/airflow/logs/download_rocket_launches/notify/2020-12-17T21:15:02.613390+00:00/1.log
[2020-12-17 21:15:30,917] {taskinstance.py:826} INFO - Dependencies all met for <TaskInstance: download_rocket_launches.notify 2020-12-17T2
[2020-12-17 21:15:30,923] {taskinstance.py:826} INFO - Dependencies all met for <TaskInstance: download_rocket_launches.notify 2020-12-17T2
[2020-12-17 21:15:30,923] {taskinstance.py:1017} INFO -
--------------------------------------------------------------------------------
[2020-12-17 21:15:30,923] {taskinstance.py:1018} INFO - Starting attempt 1 of 1
[2020-12-17 21:15:30,923] {taskinstance.py:1019} INFO -
--------------------------------------------------------------------------------
[2020-12-17 21:15:30,931] {taskinstance.py:1038} INFO - Executing <Task(BashOperator): notify> on 2020-12-17T21:15:02.613390+00:00
[2020-12-17 21:15:30,933] {standard_task_runner.py:51} INFO - Started process 1483 to run task
[2020-12-17 21:15:30,937] {standard_task_runner.py:75} INFO - Running: ['airflow', 'tasks', 'run', 'download_rocket_launches', 'notify', '2
[2020-12-17 21:15:30,938] {standard_task_runner.py:76} INFO - Job 6: Subtask notify
[2020-12-17 21:15:30,969] {logging_mixin.py:103} INFO - Running <TaskInstance: download_rocket_launches.notify 2020-12-17T21:15:02.613390+0
[2020-12-17 21:15:30,993] {taskinstance.py:1230} INFO - Exporting the following env vars:
AIRFLOW_CTX_DAG_OWNER=airflow
AIRFLOW_CTX_DAG_ID=download_rocket_launches
AIRFLOW_CTX_TASK_ID=notify
AIRFLOW_CTX_EXECUTION_DATE=2020-12-17T21:15:02.613390+00:00
AIRFLOW_CTX_DAG_RUN_ID=manual__2020-12-17T21:15:02.613390+00:00
[2020-12-17 21:15:30,994] {bash.py:135} INFO - Tmp dir root location:
 /tmp
[2020-12-17 21:15:30,994] {bash.py:158} INFO - Running command: echo "There are now $(ls /tmp/images/ | wc -l) images."
[2020-12-17 21:15:31,002] {bash.py:169} INFO - Output:
[2020-12-17 21:15:31,006] {bash.py:173} INFO - There are now 2 images.
[2020-12-17 21:15:31,006] {bash.py:177} INFO - Command exited with return code 0
[2020-12-17 21:15:31,021] {taskinstance.py:1135} INFO - Marking task as SUCCESS. dag_id=download_rocket_launches, task_id=notify, execution
[2020-12-17 21:15:31,037] {taskinstance.py:1195} INFO - 0 downstream tasks scheduled from follow-on schedule check
[2020-12-17 21:15:31,070] {local_task_job.py:118} INFO - Task exited with return code 0
```

The logs are quite verbose by default but display the number of downloaded images in the log. Finally, we can open the /tmp/images directory and view them. When running in Docker, this directory only exists inside the Docker container and not on your host system. You must therefore first get into the Docker container: