Apache Airflow - Practical Exercise

Learning objectives

In this practical exercise, you'll aim to:

- set up, run, and interact with an Airflow server;
- implement a simple Airflow DAG representing a given workflow;
- become familiar with the practical form and contents of an Airflow DAG script;
- understand how to navigate Airflow's UI; and
- be able to extend a given Airflow DAG according to new specifications.

Table of contents

To meet your learning goals, you'll work through the following sections:

1) Scenario introduction and motivation

Back to table of contents

Now that you have a basic understanding of Apache Airflow's ability to form and execute pipelines, let's develop your skills by implementing a practical workflow based on an example scenario.

Linda, your data engineering work colleague, loves kittens. In fact, there is nothing that brings her more joy than seeing pictures of cute, cuddly felines on her desktop as she codes away. Always wanting to increase her library of available kitten wallpapers, Linda has developed a workflow using the Unsplash API and a Python script to retrieve new kitten images every day.

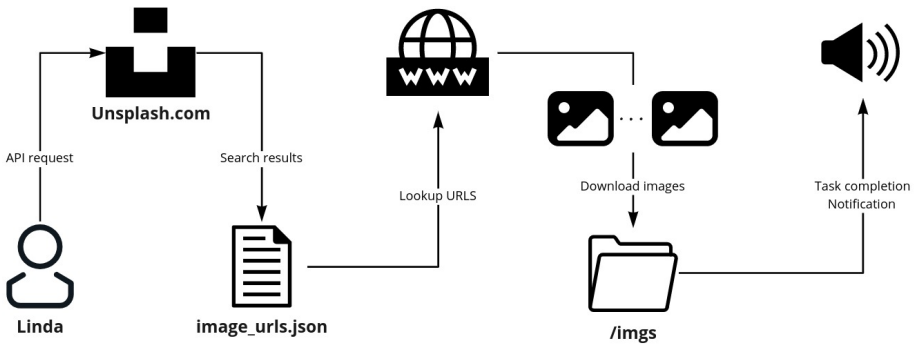

Figure 1: Linda's current kitten wallpaper workflow.

Linda's workflow, visualised in Figure 1, is a linear progression of steps:

1. She initially queries the Unsplash API searching for 'kitten' images.
2. Next, she saves the resulting API response, which contains URLs to matching images, as a JSON file on her local machine.
3. The JSON file is then parsed using a script, with each image URL being used to download an image onto her local computer.
4. The script then prints out a message once all images have been downloaded.

While the above workflow is simple and easy to implement, it possesses several downsides when you consider it as a workflow that needs to be run on a reoccurring basis (you can never have too many pictures of kittens...):

- The workflow needs to be **run manually**. There is no current functionality to schedule its execution regularly. This means that Linda has to physically trigger the workflow, or she has to set up an additional cron-like tool for its runs to be automated.

- The workflow is **not robust to failures**. Because there's only one Python script that runs the workflow, any step that fails causes the entire script to fail, meaning all the steps need to be run again. This may not seem like a big deal when you're fetching kitten images, but imagine you were retrieving and moving *terabytes* of image data â€“ re-running any step would become computationally expensive.
- In its current form, the workflow also has an**inability to scale**. Here, without the use of parallelism or the implementation of a distributed system (both of which can be non-trivial to set up), the workflow can only be run in an automated way on a single machine.
- The current approach also offers **low observability**. Unless Linda implements a large amount of logging functionality, it will not be easy for her to see why a given stage of her workflow failed, especially if this is to be observed over a number of workflow runs.

Luckily for Linda, you're around. As you've recently learned about Apache Airflow, you can explain to her how Airflow is able to:

- **provide sophisticated scheduling capabilities** that allow workflows to be run with automated schedules with even more flexibility than standard cron-based tools;
- **robustly handle workflow failures** by containing errors at a task level â€“ this means that if a given step of a workflow fails, the states of all other steps are preserved, and only the failed step is re-run (either manually or automatically according to predefined settings);
- **scale arbitrarily** as Airflow can run with multiple worker processes either on a single machine or across vast distributed systems; and
- **provide high workflow observability** through detailed, task-level logging and a rich user interface (UI) to manage and inspect multiple workflows from a single location.

As an exercise for this train, let's see how you could optimise Linda's workflow using Airflow. You've learned before that Airflow abstracts units of work into *tasks*. To form a pipeline, you define a logical dependence between these tasks, allowing the output of one or more tasks to flow into the input of a downstream task.

â± Exercise â±

Using the knowledge of tasks and DAGs that you've gained so far, consider how you would represent Linda's workflow within Airflow. Take a moment to sketch out a DAG, detailing the various tasks and the flow of dependence between them.

So, how did the exercise go? Having given the challenge some thought, hopefully you would have found yourself coming up with several possible scenarios. Examples here could include creating a DAG using only a single task, where all of the actions required to download the kitten images are performed with a single operator. Alternatively, you may have thought of a pipeline that consists of five tasks, with each one representing an arrow/transition within the original workflow. The truth is that there are many ways to express Linda's workflow as a pipeline in Airflow, with no single approach being definitively correct. There are, however, some best practices that you should apply when creating your tasks to ensure that you take full advantage of Airflow's powerful functionality. Two of these important practices see your tasks being given the properties of *atomicity* and *idempotency*:

Atomicity

Atomicity defines how a failed task should not leave your pipeline in a partially progressed state. When an atomic task fails, the state of the workflow should remain as though the task was not attempted or performed. Conversely, a completed atomic task should fully update the state of the workflow.

As an example, if a task was responsible for moving files from one folder to another, then an atomic version of the task would either move all of the files upon its successful completion or none of the files upon failure. In no case should files be partially copied between the source and destination folders.

Idempotency

If a task is *idempotent* it produces the same result (or workflow state) regardless of the number of times it is run.

As an example, consider a task responsible for writing the results of some process to a file. If this task*appended* the results to the end of the file, it would not be considered idempotent, as each time the task runs it produces a new file state (the file's contents grow larger). However, if the task *overwrites* the file each time it is executed then it would be idempotent, as it would produce an identical file output each time it runs.

The properties of atomicity and idempotency help you build pipelines that are robust to system failures and they're reproducible. Both of these characteristics are extremely important for the massive scale at which data pipelines often operate.

With the above principles in mind, we've formulated a simple three-task DAG as shown in Figure 2 below.

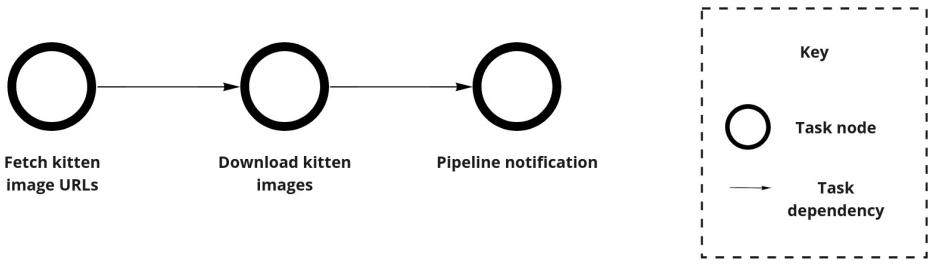

Figure 2: The workflow resolved into an Airflow-implementable DAG.

Looking at the DAG represented above you can see that the three tasks are responsible for fetching and recording the Unsplash-based kitten image URLs, downloading the URLs as images, and printing a notification message respectively.

In the following section you'll get your hands dirty by setting up your own Airflow environment, after which you'll study these tasks programmatically to see how they operate and fulfil the requirements of atomicity and idempotency.

2) Initial Airflow environment setup and configuration

Back to table of contents

In this section we'll describe the environment you can use to run Airflow and your implementation of Linda's workflow. We'll also provide configuration steps which you need to complete to run the workflow pipeline.

2.1) Environment overview

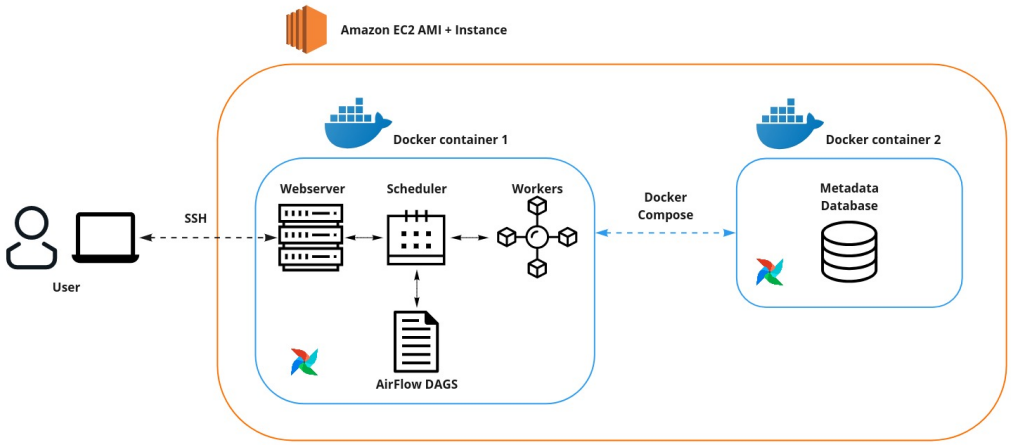

Figure 3: The environment setup used to run Apache Airflow for this exercise.

Airflow is a sophisticated piece of software that when running in production-scale environments can be deployed across hundreds of machines in a distributed manner. This means that unfortunately there is a fair amount of complexity involved in its setup, even if you're wanting to simply experiment with its basic functionality. For this reason, we've provided a pre-configured Amazon EC2 AMI that has all the dependencies required to run this train.

Visualised in Figure 3, the AMI uses theDocker platform to maintain two Airflow containers that host the UI, scheduler, workers, and DAG script(s), as well as a PostgreSQL-based metastore database respectively. These two containers are initialised via the Docker Compose tool and work in tandem to provide the overall Airflow functionality. A user can access the Airflow UI by remotely logging into the AMI (using `ssh` and a security key generated within EC2) and forwarding the Airflow UI port (8080) to their local machine. This process is described in Section 2.3.

2.2) Creating an Unsplash API key

To successfully implement your workflow within Airflow, you'll need access to the Unsplash API from where you can retrieve images of the cutest and cuddliest kittens ever (we're really sorry if you're a dog person ðŸ•). The brief instructions below will guide you through the process to set up a free developer account with Unsplash to access their API:

1. Navigate to the Unsplash Developer site. Click on "Register as a developer".
2. Provide the requisite information to create an Unsplash account. You should now be redirected to your home account page, similar to Figure 4.

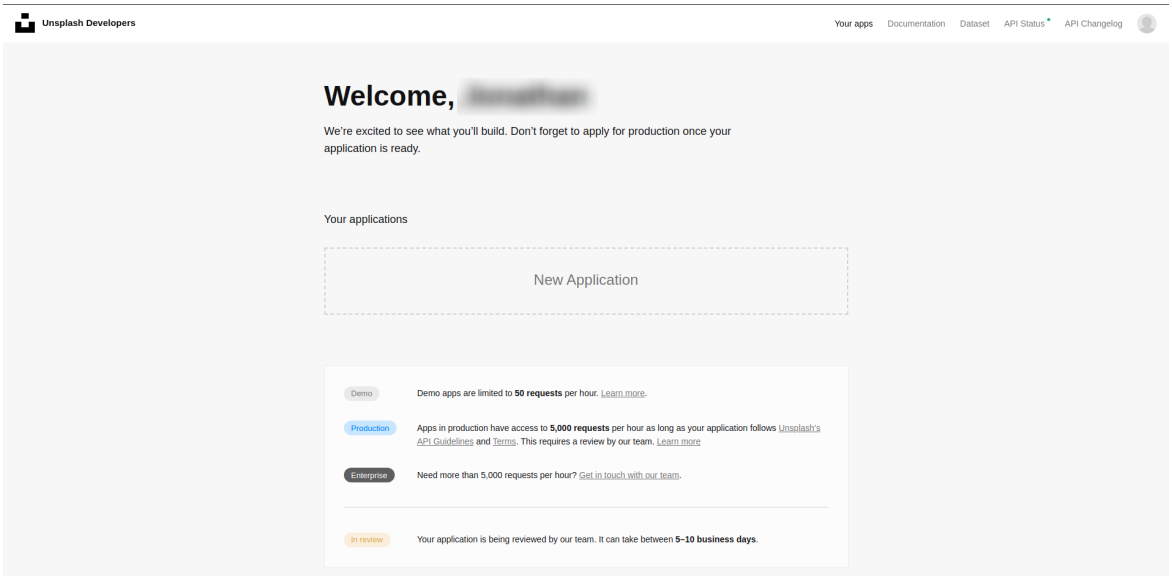

Figure 4: The overview screen of a newly created Unsplash account.

3. Click on "New Application". Accept the API Use and Guidelines terms.
4. Provide a name and brief description for the application, such as:
   - Name: *'Airflow_basic_pipeline'*
   - Description: *'Airflow pipeline to extract kitten images from Unsplash.'*
5. Scroll down to the middle of your application page to retrieve your API key stored in the `Access Key` field (shown in Figure 5). Keep this key in a secure location to be used in Step 3.3.

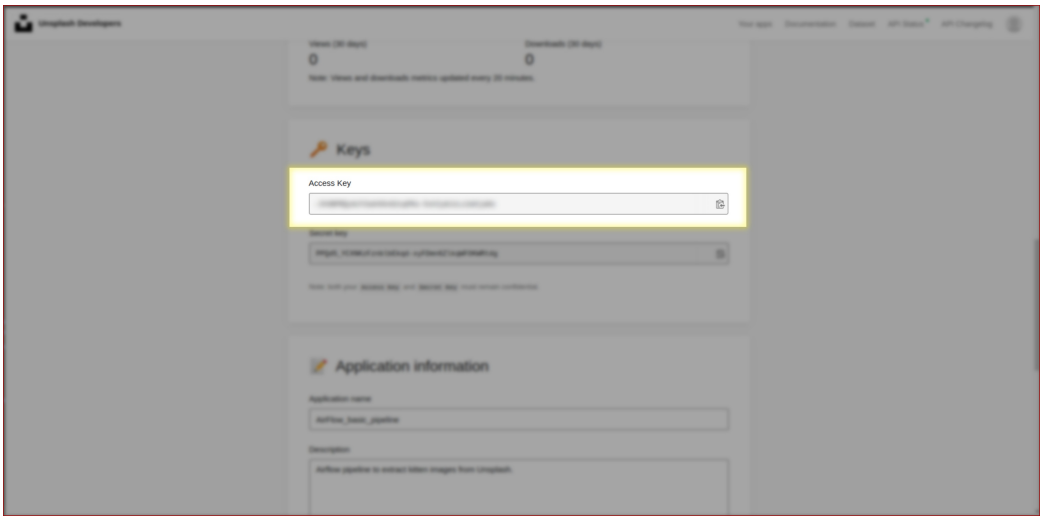

Figure 5: The Unsplash API access key is found roughly halfway down the dev account page.

And there you go â€“ the API setup is complete!

2.3) Launching the Airflow environment in AWS

Following on from its description in Section 2.1, the following steps set up the Airflow environment in which you can implement a pipeline solution to Linda's kitten image workflow.

1. Sign in to the EC2 dashboard via the AWS management console, and begin the process of launching a new EC2 instance.
2. When prompted to select an AMI, select the "Community AMIs" tab, and search for *"edsa-airflow-basic"*. Select the resulting AMI image.
3. In the following step when prompted to choose an instance type, select a `t2.small` instance.
4. Proceed normally through the rest of the setup process ensuring that the instance has EBS storage provisioned (8 GB is sufficient) and appropriate security group permissions set up (`ssh` access is only required).
5. When launching the instance, make sure you select or create a key pair that you have easy access to.
6. Complete the initial AMI configuration by selecting "Launch".
7. With your instance launched, navigate to the *EC2 Instances* view and find the public IP of your machine. Store this value in a convenient location to use in the next step.
8. Log into the remote instance using the following command:

   ```
   ssh -i {you-keypair-name}.pem -L 8080:localhost:8080 ubuntu@{your-ec2-instances-public-IP}
   ```

   By now, most of this command should look familiar to you. It is used to set up a secure shell connection with the remote EC2 instance. One new parameter that you may have noticed is the `-L 8080:localhost:8080` flag. This parameter allows you to perform *'port forwarding'*, which essentially means that content that is served on the remote instance's port `8080` is also transferred to your local computer's port `8080`. This allows you to view the Airflow UI locally, even if it is actually being hosted on the remote instance.
9. Within the instance, navigate to the root of the Airflow exercise folder: `/home/ubuntu/EDSA-airflow/initial_pipeline_creation`.
10. Launch the Airflow stack by entering the command:

    ```
    docker-compose up
    ```

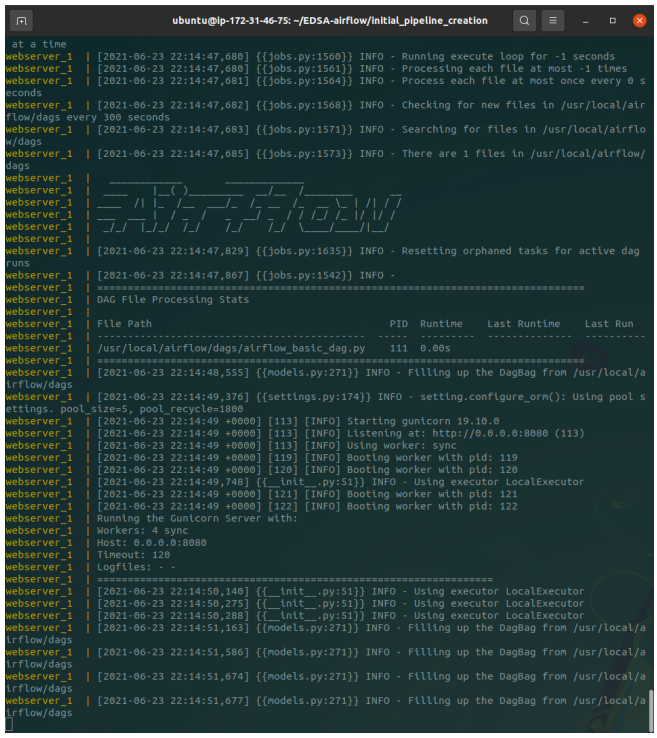

Figure 6: Once the 'docker-compose up' command is run, a large amount of logging text should appear in the terminal, indicating the Airflow docker containers are setting themselves up. After the Airflow logo is printed (as seen above), the Airflow service will be ready for operation.

11. If followed correctly, you should see a long stream of output within your terminal from the `webserver_1` and `postgres_1` docker containers.
12. Test that the setup is working correctly by navigating to `localhost:8080` within a browser of your choice on your local machine. Here, if successful, you should be presented with the overview screen of the Airflow UI (seen in Figure 7 below).

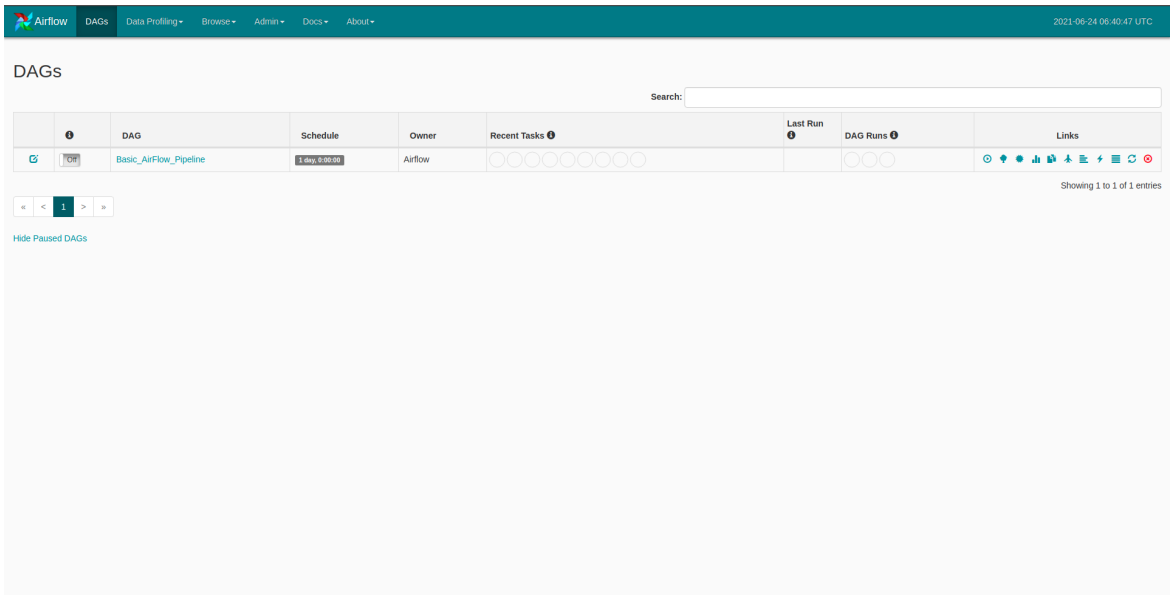

Figure 7: The Airflow UI's overview page.

3) DAG overview and configuration

Back to table of contents

Having successfully launched the Airflow service, let's consider how Linda's simple workflow (presented in Figure 2) is represented using an Airflow DAG file. Fortunately, you already have a fully implemented version of this file within your environment, and you'll consider each of its logical parts in the subsections below.

To see and edit the complete file you can either open it in a second terminal window (ssh into your instance a second time and open the file found at `/home/ubuntu/EDSA-airflow/initial_pipeline_creation/dags` using your favourite terminal text editor, such as `nano` or `vim`), or you can navigate to the DAG code overview page of the Airflow UI by clicking on the exercise DAG (`Basic_AirFlow_Pipeline`) and selecting the `Code` tab (as indicated in Figure 8).

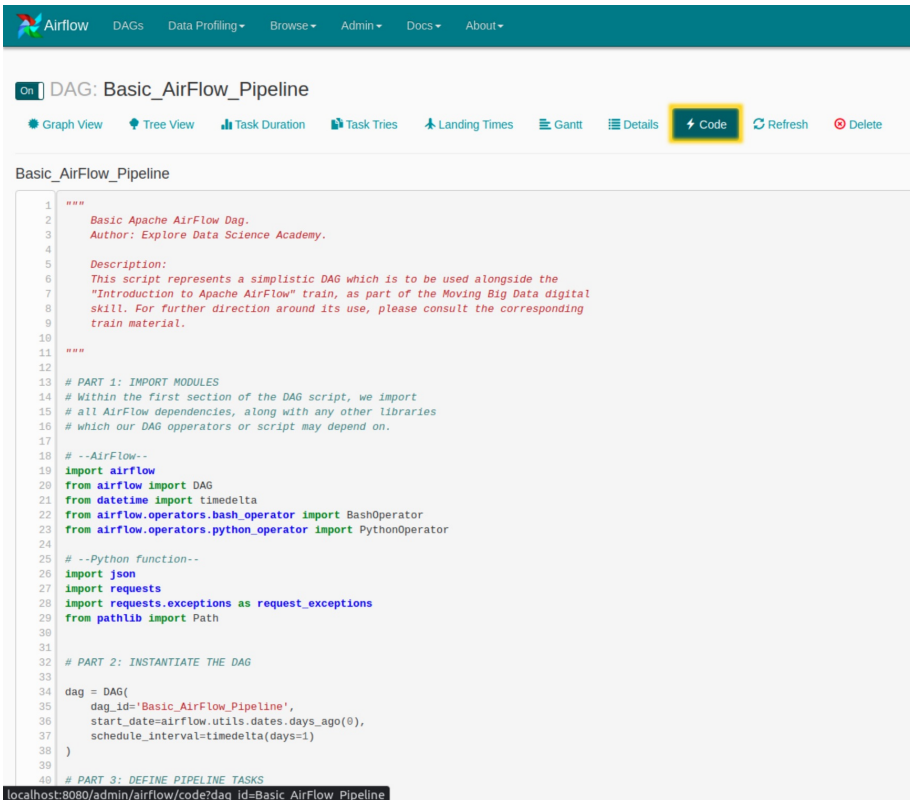

Figure 8: The DAG code overview page within the Airflow UI.

With the code in front of you, let's review its logical parts. As you've learned before, although the contents of these parts will vary based on the pipeline you're forming, the general structure of a DAG will conform to this same pattern.

3.1) Module imports

Initially, your DAG file begins by defining the module dependencies required for its execution within Airflow, along with Python libraries and modules that your individual tasks may utilise.

```
# --Airflow--
import airflow
from airflow import DAG
from datetime import timedelta
from airflow.operators.bash_operator import BashOperator
from airflow.operators.python_operator import PythonOperator

# --Python functions--
import json
import requests
import requests.exceptions as request_exceptions
from pathlib import Path
```

When looking at the Airflow-related imports, you can initially see the presence of the `DAG` class. This class allows you to instantiate a DAG object that you continuously configure throughout the script, and which ultimately becomes the definition for your entire workflow/pipeline in Airflow. As a core feature of Airflow is the facilitation of scheduled pipeline runs, you can use the `timedelta` class to help define the frequency at which your pipeline should be run. You can also see two `Operator` dependencies being imported, namely the `BashOperator` and `PythonOperator` respectively. As you've learned before, operators allow you to define the fundamental tasks which need to be run in Airflow, with each implemented operator representing a task in your DAG. You'll soon see how to use these operators to build up the various tasks within your simple workflow for Linda.

Looking at the 'Python functions' section of the imports, note the number of libraries that help you connect to the Unsplash API and the resulting image data it provides.

3.2) DAG instantiation

With your DAG dependencies imported, you can now instantiate your script's DAG object.

```
dag = DAG(
    dag_id='Basic_Airflow_Pipeline',
    start_date=airflow.utils.dates.days_ago(0),
    schedule_interval=timedelta(days=1)
)
```

Here, upon creation, you pass a couple of arguments for configuration:

- `dag_id`: This is essentially the name that Airflow will use to track your DAG.
- `start_date`: With its powerful scheduling abilities, this field is used by the Airflow scheduler to indicate when your pipeline should begin to run. Here, it may seem strange that you should define the start date with a `days_ago()` function. After all, surely your pipeline can only start executing at the present or some future date, right? As it turns out, this is actually an intentional feature of Airflow, and it enables a defined pipeline to perform *backfilling*. While a full exposition of this functionality is beyond the scope of this train, simply put, backfilling allows you to run your pipeline as though it existed in the past, ingesting and processing historical data seamlessly. This feature is extremely useful for cases where your pipeline may have been updated and needs to reflect this update in the processing of not only new but past data as well.
- `schedule_interval`: Given a start date, the schedule interval defines how frequently a pipeline should run. This parameter is extremely flexible and accepts both cron-based time intervals or more nuanced `timedelta` intervals as provided above.

3.3) Pipeline task definitions

Let's now turn your attention to the real work of the pipeline: the individual tasks defined within your workflow. As noted before, each task is defined within an Airflow operator object that you instantiate. All of these operators derive from a common `BaseOperator` class, and it's great to know that you can create your own operators if you ever felt compelled to do so (Airflow has an abundance of community-created operators, so be sure to do some searching before you decide that you need to create your own).

In reviewing your tasks, you'll locally progress along Linda's workflow from start to finish:

Fetch kitten URLs

âš¡ Input required âš¡

For your workflow to run correctly, you'll need to add the details of your Unsplash API access key, as obtained in Step 2.2, within the `CLIENT_ID` variable.

```
CLIENT_ID = '{Your-Unsplash-Access-key-here}'

fetch_kitten_urls = BashOperator(
    task_id="fetch_kitten_urls",
    bash_command=f"curl -o /tmp/images_urls.json -L \
             --request GET 'https://api.unsplash.com/photos/random?query=kitten&count=2&client_id={CLIENT_ID}'",
    dag=dag
)
```

As mentioned before, the first task in your pipeline is to make a call to the Unsplash API and retrieve some adorable kitten images. Unsplash does this by providing a JSON object in response to your query, which takes the form of a `GET` request within which you pass your query string (`query=kitten`), the number of images you want to receive (`count=2`), and your API access key (`client_id={CLIENT_ID}`). Note that for your request to execute successfully, you'll need to update the `CLIENT_ID` variable to contain your unique Unsplash API access key, as obtained in Step 2.2.

As the process of making your API call and receiving the corresponding JSON file can all be performed via the command line, you can create your first task using the `BashOperator`, which allows Airflow to execute a specified shell command.

Use the following parameters when configuring this operator:

- `task_id`: In a pattern that you'll see throughout the remainder of the other operators and tasks, use this parameter to give your task a unique name. A common practice here is to give the `task_id` and the resulting task operator object the same name (in this case `fetch_kitten_urls`).
- `bash_command`: Intuitively, this command allows a user to specify a bash command that the task should execute. In this case, use the `curl` command-line tool to send your `GET` request to the Unsplash API server (`https://api.unsplash.com/`) and to store the resulting JSON file (`mages_urls.json`) in the `/tmp/` folder. While beyond the scope of this exercise, it's good to understand that Airflow supports dynamic templating, which allows you to pass in unique values to your bash operator every time your pipeline is run in Airflow. For example, this means that you could give the JSON file that you download a unique name every time the pipeline is run, based on the run number.
- `dag`: This parameter is used to associate your task with the DAG object you created before. Omitting this line will result in your task not being captured within your overall workflow.

Get kitten images

You've now arrived at the most involved task within your pipeline, which is responsible for unpacking the `images_urls.json` file you captured in the previous task and downloading the contained image URLs as image content onto your computer. While you could perform this action using bash commands, doing so would be a complex feat, and it's a problem that is far easier to solve robustly in Python. As such, for this second task, make use of the `PythonOperator`. Unlike the `BashOperator` that you used before, using this operator requires two components. You initially need to define a Python function to execute, after which you need to instantiate the `PythonOperator` object itself and pass your created function's signature to it. The `PythonOperator` is best used when running small code snippets and dedicated Python functions instead of complete scripts. For intricate or more complicated Python code, the `BashOperator` gives better performance.

Let's look at these two components below, starting with your defined function:

```
def _get_kitten_images(json_file: str, save_path: str):
    """Extracts and stores images fetched from the Unsplash
       image API.

    Args:
       json_file (string): Path to the output json file from the Unsplash
                           API request.
       save_path (string): Path to save retrieved images.
    """
    # Load the contents of the JSON file.
    with open(json_file, 'r') as jsn_file:
       api_data = json.load(jsn_file)

    # Initially validate that all supplied URLs operate correctly,
    # saving the corresponding image data within an array.
    img_data_array = []
    for item in api_data:
       try:
          img_data_array.append(requests.get(item["urls"]["small"]).content)
       except request_exceptions.MissingSchema:
          print (f'{item["urls"]["small"]} appears to be a malformed URL')
       except request_exceptions.ConnectionError:
          print (f'Could not retrieve URL at: {item["urls"]["small"]}')

    # Generate save directory if needed.
    save_path = Path(save_path)
    save_path.mkdir(exist_ok=True)

    # Save the gathered image data as images.
    for i, item in enumerate(api_data):
       img_path = save_path.joinpath(f"{item['alt_description'].replace(' ', '-')}.png")
       with open(img_path, 'wb') as img_file:
          img_file.write(img_data_array[i])
          print (f'Downloaded {item["urls"]["small"]} to {img_path}')
```

While we won't perform a line-by-line commentary of the above function (we encourage you to read through it and understand its contents), there are some high-level points to capture:

- The general flow sees the JSON file being initially loaded, whereafter each image URL is parsed from the JSON data and is attempted to be fetched. If successful, the resulting image data collected is then saved into memory as images at a location defined by the `save_path` parameter of the function. Take a moment to think about how this function structure allows the task to be both *atomic* and *idempotent*.
- Note that the function takes in two arguments: both the location of the JSON file and the folder in which to store the retrieved images.
- Name your function with a leading underscore (`_get_kitten_images()`) to indicate that it'll be used within a task operator.

With your function defined you can now see how to include it within your pipeline through your `PythonOperator`:

```
get_kitten_images = PythonOperator(
    task_id="get_kitten_images",
    python_callable=_get_kitten_images,
    op_kwargs={
        "json_file": "/tmp/images_urls.json",
        "save_path": "/tmp/fetched_images/"
    },
    dag=dag
)
```

In a similar pattern to your previous task, when instantiating your `PythonOperator` object there are several parameters that you need to specify:

- `task_id`: As seen before, this uniquely identifies your task within the Airflow DAG.
- `python_callable`: Use this parameter to provide the signature of the Python function you'd like to execute. In this case, this is the `_get_kitten_images` that you defined above.
- `op_kwargs`: This parameter accepts a dictionary of key-value pairs representing the *optional keyword arguments* that should be passed to your Python function. Here, you specify paths for both the `json_file` and `save_path` parameters of your function.
- `dag`: Again, this parameter is used to associate your given task to the Airflow DAG you are building.

Pipeline notification

As a final task within your workflow, Linda would like you to provide a notification of the end-state of the pipeline whenever it completes. While there are many ways you can do this (for example, using an `EmailOperator` to send an email notification to her personal email address), for simplicity in this exercise you'll log the number of kitten images currently gathered by the pipeline. Do this using another `BashOperator` as defined below:

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

Much like your first task, the `pipeline_notification` task instantiates a `BashOperator` object with the following parameters:

- `task_id`: The unique name of your task within the DAG.
- `bash_command`: Use the `echo` command line utility, along with the word-count (`wc`) and list directory (`ls`) commands to print out the number of files in the `/tmp/fetched_images/` folder within your Airflow docker container, representing the number of kitten images you've downloaded so far.
- `dag`: Hopefully this is familiar by now, as you will use this parameter again to associate this task with your constructed DAG.

3.4) Task flow

As a final part of your DAG script you need to define dependencies between your specified tasks. You've already seen several examples of how this is performed, and within our example script we mirror the task flow represented in Figure 2.

```
fetch_kitten_urls >> get_kitten_images >> pipeline_notification
```

4) Pipeline triggering and inspection

Back to table of contents

With your environment set up and an understanding of your DAG for Linda's workflow, let's go ahead and run the pipeline!

4.1) Accessing the Airflow UI

As explained in Section 2.3, the Airflow UI should be accessible by simply navigating to `localhost:8080` within an internet browser of your choice.

4.2) Triggering the pipeline

To run your pipeline there are two quick actions you need to perform on the Airflow landing page. As illustrated in Figure 9, you initially need to *enable* your DAG by selecting the on/off toggle switch to the left of the screen corresponding to your DAG's name (*"Basic_Airflow_Pipeline"*). Following this, you can *trigger* the pipeline by selecting the small play button on the right of the screen.

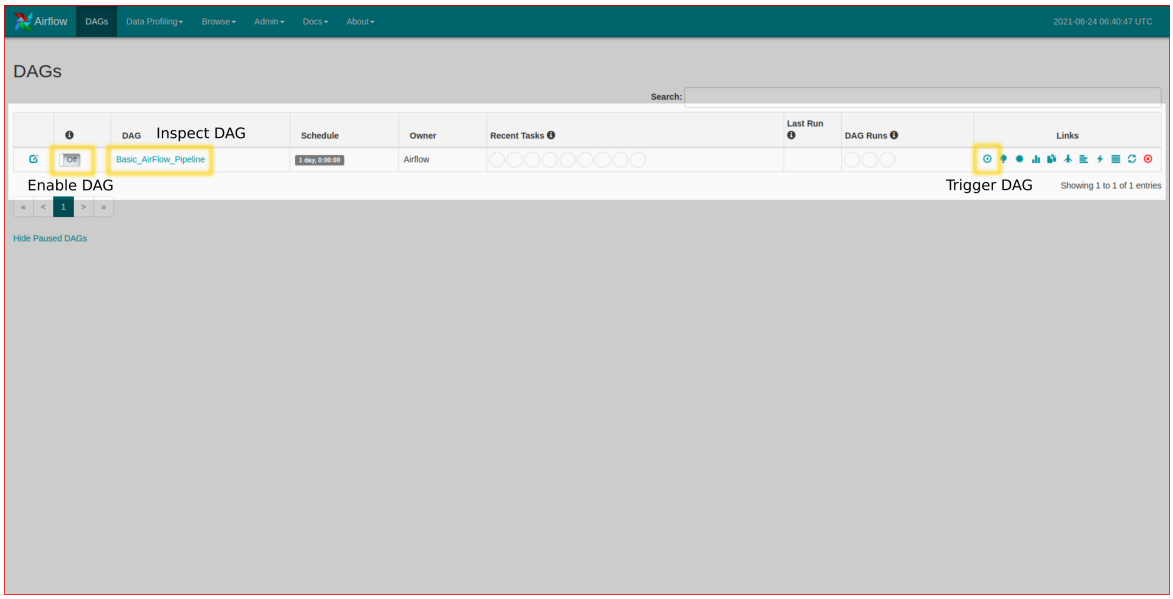

Figure 9: The Airflow UI overview page with triggering mechanisms annotated.

Once triggered, Airflow will begin humming away in the background, and you should see the terminal output indicating various operations being performed. After a couple of seconds, hit the refresh button associated with the DAG (near the trigger button), and you should see the UI indicating that the task has executed successfully, as represented in Figure 10.

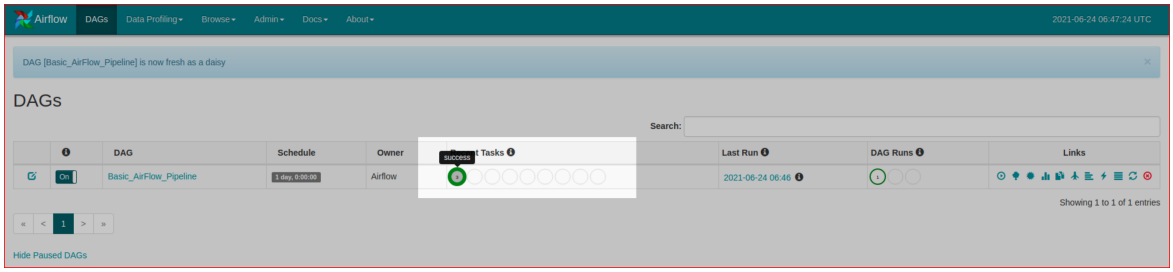

Figure 10: The Airflow UI overview page with all tasks having successfully completed.

ðŸ› Debugging your workflow ðŸ›

The first time you try and run your Airflow DAG you may disappointingly see that not all tasks complete successfully. There are two possible reasons this may occur. First, you might have forgotten to place your Unsplash API access key within the DAG definition (as instructed in Section 3). Second, Airflow may not have initialised correctly upon startup. If this is the case, the problem is often simply solved by re-triggering your pipeline to run.

4.3) Pipeline inspection

One of the powerful features of Airflow is its ability to present immense detail surrounding the performance and state of DAGs through the Airflow UI. Having successfully run your workflow, let's take a quick tour around the Airflow UI to get further insight into how your pipeline executed.

The DAG tree view

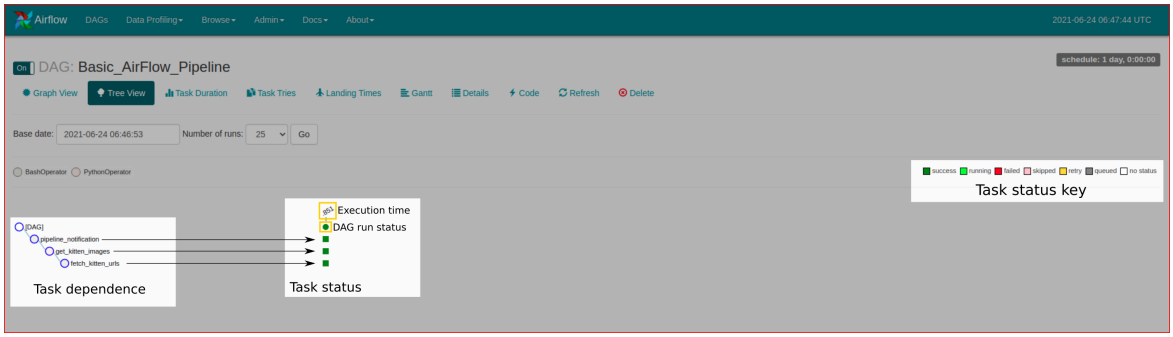

Figure 12: The tree view of the successfully completed DAG. Annotations are provided to help show the representational power of this visualisation.

The first visual you'll inspect is the tree view. You can navigate to this page by clicking on your DAG's name from the landing screen, or by selecting the *Tree View* tab when already inspecting a DAG.

The tree view provides two interrelated visuals.

On the left, a hierarchical representation of each task's dependence within the DAG is given, with each branch being executed from the leaf node (bottom of the branch) to the root node (top of the branch). The presence of multiple branches (not shown in Figure 12) represents how tasks within your DAG can run independently of one another until depending on a common task node.

On the right of the tree view, the second visual provides a growing grid-like display that increases each time your DAG is run. Here, each row of the grid represents a single task within your DAG (aligned to the tree visual). Each column of the grid represents a point in time at which the entire DAG was run. Combined, each cell within the grid represents the status of a given task at a point in time in which the entire DAG was run. This gives a powerful view of how your pipeline is performing over time and where problems may be arising due to specific task failures.

The DAG graph view

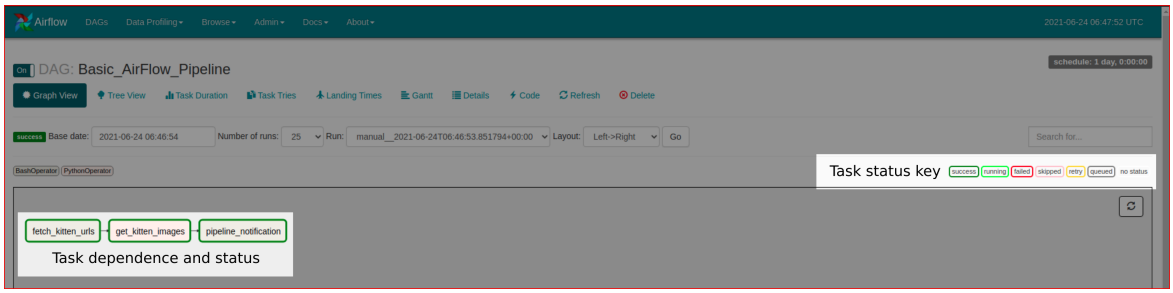

Figure 12: The graph view of the successfully completed DAG, as visualised within the Airflow UI.

By clicking on the *Graph View* tab, you can explore the next visual in the Airflow UI. Whereas the tree view helps you visualise the state of various tasks over time, the graph view provides a full representation of your DAG during its current run cycle. This lets you visually map out the status of various tasks and their downstream dependencies.

Viewing your notification task output

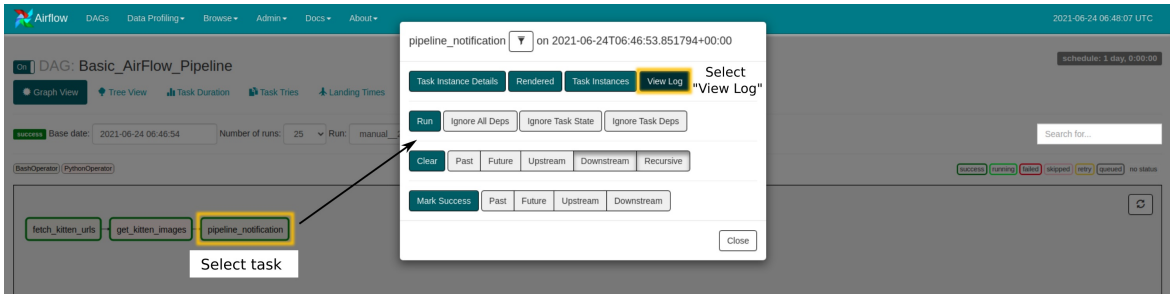

Figure 13: Navigating to a task's log data within the graph view of the Airflow UI.

While in the graph or tree views of the Airflow UI, you can also retrieve detailed information around a given task by inspecting its logs. Let's see an example of this by checking that your pipeline_notification task is indeed correctly reporting the number of kitten images you've downloaded. To do this, click on the pipeline_notification task from the graph view (or the corresponding cell within the tree view), and then select "View log". This process is represented in Figure 13.

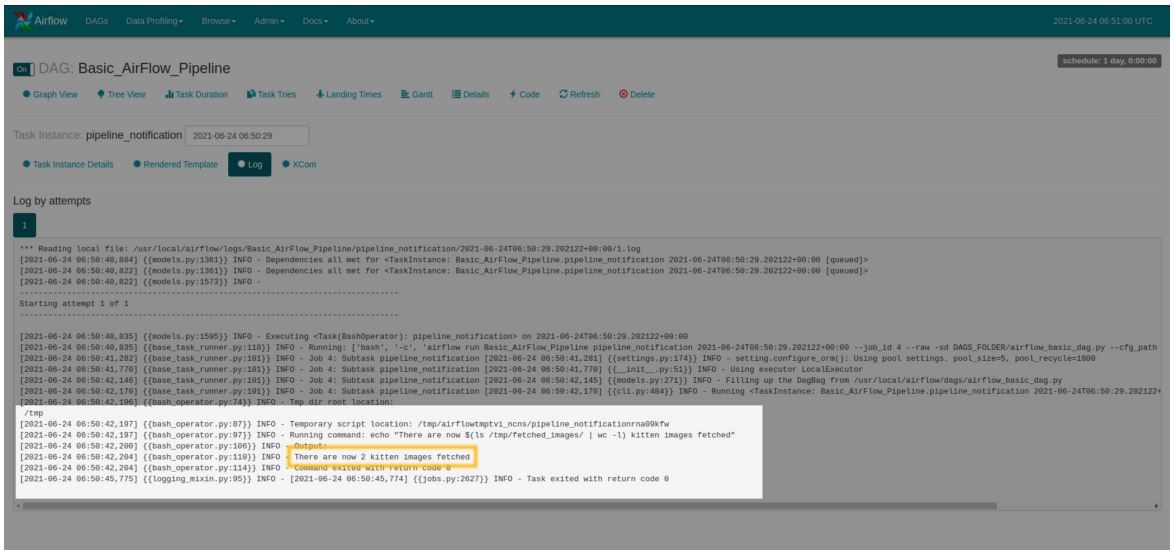

Figure 14: Viewing the log-based output of a 'pipeline notification' task within the Airflow UI.

Once you've chosen to view the task log, you should see a large amount of verbose text representing the logging information captured by Airflow. As shown in Figure 14, scrolling towards the bottom of this log should reveal the line echoed by your notification task, stating that there are a certain number of images fetched (this should be the number of times the pipeline has been run multiplied by two).

â„¹ï¸ Viewing kitten images â„¹ï¸

If you're just as excited as Linda to see the downloaded kitten images, you can navigate to the home/ubuntu/EDSA-airflow/initial_pipeline_creation/imgs folder where the downloads are stored. You'll then need to use a remote file system utility such as scp or rsync that can copy this directory onto your local machine, allowing you to display the images â€“ whiskers and all.

The ability to view task logs is extremely useful when trying to debug a large or complex workflow and is indispensable as a feature for Airflow.

Additional views

Besides the graph and tree views that you've covered, there are several other visuals which Airflow provides to help you monitor, debug, optimise, and report on your workflow. We encourage you to use your environment as a sandbox and to explore these additional visuals under the tabs, such as *"Task Duration"*, *Task Tries*, *Landing Times*, and *Gantt*.

[Exercise] Extending the pipeline

Back to table of contents

Having gained a better understanding of Airflow and its operation, it's now time for you to assess your understanding by extending Linda's current workflow. Here, Linda is keen to have not only a silent message printed out to the console when her workflow is run, but she would also like you to create a log file that can permanently capture the current state of the image folder to which her kitten images are saved.

To perform this update for Linda, you'll need to extend the current definition of the `Basic_AirFlow_Pipeline` DAG by including an additional task and its dependence within the current workflow. Here are some details to guide your implementation:

- Name the new task `write_results_to_file`.
- This task can be implemented using a simple `BashOperator`.
- Use your knowledge of Linux command line utilities to alter the command run by the `pipeline_notification` task, making it write the notification string to a file instead of to stdout.
- Your results should be saved to `/usr/local/airflow/kitten_state.txt`.
- This new task should run in parallel with the `pipeline_notification` task.

The resulting DAG with the updated task and dependence is represented in Figure 15.

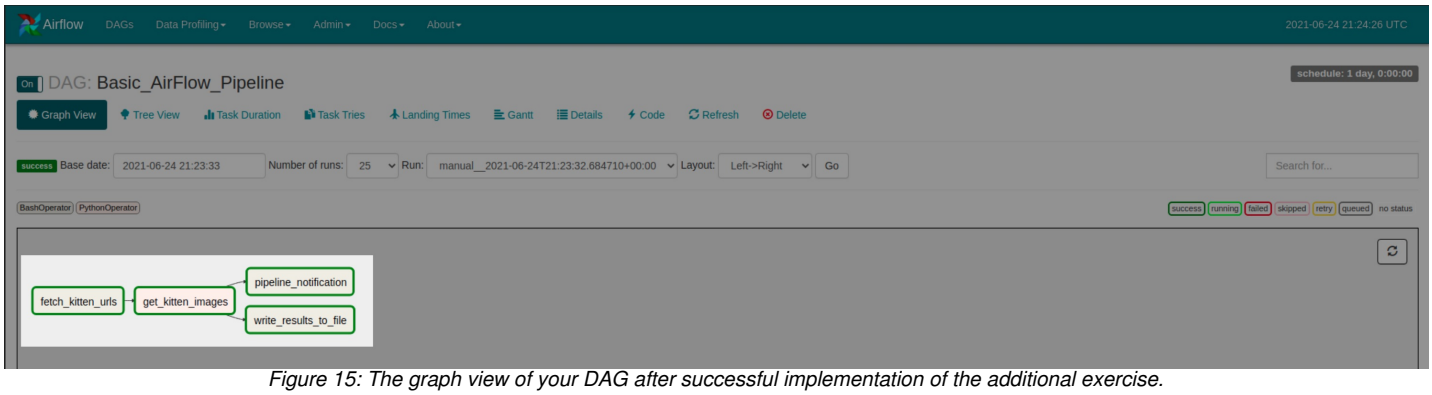

Figure 15: The graph view of your DAG after successful implementation of the additional exercise.

Conclusion

Well done! If you've made it this far and have completed the above exercise, then you're well on your way to mastering Apache's Airflow. In this practical train, you set up, ran, and interacted with an Airflow server. By studying a DAG Python script, you saw how to implement a simple Airflow DAG representing a given workflow. This allowed you to become familiar with the practical form and contents of an Airflow DAG script and led you to understand how to navigate Airflow's UI to evaluate a given pipeline's execution. Lastly, through the provided exercise, you've been able to extend a given Airflow DAG according to new specifications, gaining further understanding of the tool.

There is far greater depth to Airflow than what you've covered in this brief exercise, with entire textbooks being written on this tool. As such, we encourage you to muster the Explorer spirit and strive to increase your knowledge of this powerful piece of software.

References

- Additional Information on Configuring a DAG within Airflow
- A Guide to Templating in Airflow
- The Official Introduction Tutorial to Airflow
- Scheduling in Airflow