

# Getting started with Airflow

This guide aims to help you get started with Hogwarts Airflow. After following this guide, you should be able to create, test and deploy your very first DAG on the Hogwarts platform!

 A special thanks to Nathalie Faucher for writing the first version of this guide.

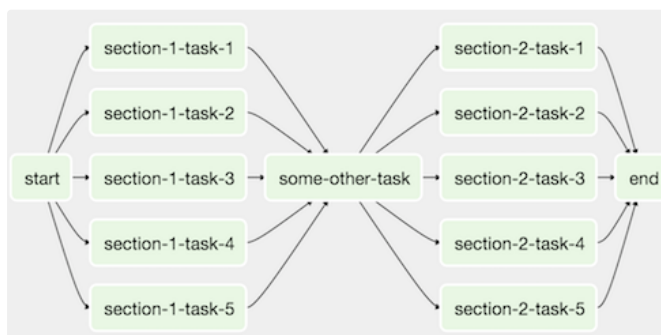
- [Vocabulary](#)
- [Creating your DAG](#)
  - [Basics](#)
- [Testing your DAG](#)
- [Deploying your DAG](#)
  - [Templates and examples](#)
  - [Internally developed Operators \(Daggers\)](#)
- [Feedback](#)
- [More information & links](#)
  - [Official docs](#)
  - [Internal docs](#)
  - [Great articles/resources](#)

## Vocabulary

Before starting on your Airflow journey, here are a few Airflow specific concepts you should know about:

### ▼ DAG (Directed Acyclic Graph)

A Python configuration file used to represent a collection of all tasks you want to run, organized in a way to reflect their relationships and dependencies. [Official Airflow definition](#)



### ▼ DAG run

An instance of a DAG, containing task instances that run for a specific `execution_date`.

DAG runs can be created on a schedule interval or triggered manually.

[Official Airflow docs - DAG model](#)

### ▼ Task

Defines a unit of work within a DAG; it is represented as a node inside the graph.

Each task is defined by an operator, which will be instantiated at run time. The work done by this task will depend on the implementation of the operator (e.g. `PythonOperator` will run Python code).

Tasks can have dependencies between each other to allow for better planning and control of the execution flow. These dependencies are constructed by using the *upstream* `<<` and *downstream* `>>` operators.

Here is an example from the Airflow docs:

```
1 with DAG('my_dag', start_date=datetime(2016, 1, 1)) as dag:
2     task_1 = DummyOperator('task_1')
3     task_2 = DummyOperator('task_2')
4     task_1 >> task_2 # Define dependencies
```

Using the code above, we can say that `task_1` is *upstream* of `task_2`, and conversely `task_2` is *downstream* of `task_1`.

#### ▼ Task instance

An instance of a task. This instance is where the actual work is being done.

[Official Airflow docs - Task instance model](#)

#### ▼ Execution date

The logical date and time at which a DAG run and its task instances are running for.

The `execution_date` can be thought of as the *start* of a schedule interval. It is also static and will not change if a DAG run or task instance is re-ran multiple times.

#### ▼ Start date

The date and time defined at the DAG level (or *default\_args*). This date and time represents when a DAG run and its task instances can start being scheduled.

**Note:** There also exists `start_date` and `end_date` at the task instance level. These dates are different and solely represent the actual date and time at which a task instance started and completed its execution.

## Creating your DAG

### Basics

The basics for creating your initial DAG Python file are the following:

#### ▼ 1. Import modules

As typical for all Python files, the DAG file should start with the imports.

```
1 from datetime import datetime, timedelta
2
3 from airflow import DAG
4 from airflow.operators.dummy_operator import DummyOperator
5 from airflow.operators.python_operator import PythonOperator
```

#### ▼ 2. Create a default\_args dictionary

A default arguments dictionary. This object should contain default values you wish to pass to all tasks. Some arguments may be required (i.e. `owner`, `start_date`)

```
1 default_args = {
2     'owner': '<owner>', #required
3     'start_date': datetime(2021, 4, 21), # required - but can also be passed at DAG level
```

```

4     'email': '<email>',
5     'email_on_failure': False,
6     'email_on_retry': False,
7     'retries': 1,
8     'retry_delay': timedelta(minutes=5)
9 }

```

### 3. Instantiate your DAG

Create the DAG instance by specifying a *unique* `dag_id`, passing the `default_args` from **step 2**, and specifying a `schedule_interval`. More parameters can be used to customize your DAG execution (e.g. `catchup`, `concurrency`, `max_active_runs`, etc.)

```

1 dag = DAG(
2     dag_id='hello_world',
3     description='A DAG which runs every 5 minutes',
4     default_args=default_args,
5     schedule_interval='5 * * * *',
6     is_paused_upon_creation=False,
7     catchup=False
8 )

```

### 4. Create your task instances

Create all of your task instances to represent the work to be done. These should reference the DAG created in **step 3**, and use a *unique* `task_id` (unique to the DAG).

```

1 dummy_task = DummyOperator(
2     dag=dag,
3     task_id='dummy_task'
4 )
5
6 def print_hello():
7     return 'Hello world of people interested in airflow :)'
8
9 python_hello_task = PythonOperator(
10     dag=dag,
11     task_id='python_hello_task',
12     python_callable=print_hello,
13     retries=3
14 )

```

### 5. Set up your task dependencies


Finally, define your execution flow by creating the necessary dependencies between tasks using the upstream `<<` and downstream operators `>>`.

```

1 dummy_task >> python_hello_task

```

With the execution flow above, the `dummy_task` task will run first. Once it completes successfully, its downstream tasks will be executed (i.e. `python_hello_task`). The DAG run will complete once all tasks have completed.

 A `DummyOperator` task won't run any code. It is simply a *dummy* node within your Directed Acyclic Graph that can be used to group up task dependencies.

# Testing your DAG

Here is a step-by-step guide on how to test your DAG on a JupyterHub server:

- 1. Create your own JupyterHub server in the same environment you expect it to be scheduled in (U, PB, etc.)

JupyterHub servers come with a pre-installed version of Airflow. This version will match the one used on the Hogwarts Airflow platform.

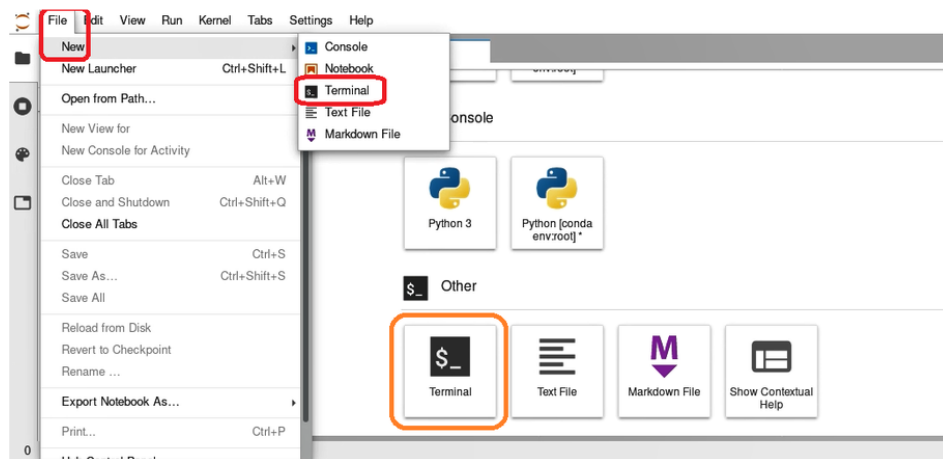
Furthermore, JupyterHub servers and Airflow workers use the same Docker image to run your workload. This makes it easy to replicate the production environment in which your Airflow tasks will run.

Links to our JupyterHub environments:

- U (dev): <https://jupyterhub.hogwarts.udev.azure.chimera.cyber.gc.ca>
- U (prod): <https://jupyterhub.hogwarts.u.azure.chimera.cyber.gc.ca>
- PB (prod): <https://jupyterhub.hogwarts.pb.azure.chimera.cyber.gc.ca>

- 2. Start a terminal within your JupyterHub server

Once logged into your JupyterHub server (Pyspark), create a terminal session:



- 3. Initialize the Airflow database

Run the following command to initialize Airflow's local SQLite database:

```
1 airflow db init
```

⚠ If you encounter this error: `AttributeError: columns`, please pin your `sqlalchemy` version to 1.3.24 by running `pip install sqlalchemy==1.3.24` to resolve the issue.

- 4. Add your DAG(s) to the /dags folder

Once initialized, Airflow will set up shop under your home directory as the following folder: `~/airflow`. In that folder, create a folder named `dags` - this is where the Airflow scheduler will source DAGs to add or update into the database.

📁 As a good exercise, add the example DAG we built in the 'Create your DAG' section!

- 5. Run the Airflow scheduler

**In terminal**

To run the Airflow scheduler in your current terminal session, run the following command:

```
1 airflow scheduler
```

If you added the example DAG in the previous step, you should see something like this (can take up to 1-2 mins):

```
jovyan@jupyter-mesard-5sebastien:~$ airflow scheduler
[2021-05-03 18:56:04,221] (scheduler_job.py:1247) INFO - Starting the scheduler
[2021-05-03 18:56:04,222] (scheduler_job.py:1252) INFO - Processing each file at most -1 times
[2021-05-03 18:56:04,226] (dag_processing.py:250) INFO - Launched DagFileProcessorManager with pid: 6651
[2021-05-03 18:56:04,228] (scheduler_job.py:1834) INFO - Resetting orphaned tasks for active dag runs
[2021-05-03 18:56:04,246] (settings.py:54) INFO - Configured default timezone Timezone('UTC')
[2021-05-03 18:56:04,260] (dag_processing.py:515) WARNING - Because we cannot use more than 1 thread (parsing_processes = 2 ) when using sqlite. So we set parallelism to 1.
[2021-05-03 18:56:21,105] (dagrun.py:445) INFO - Marking run <DagRun hello_world @ 2021-05-03 17:05:00+00:00: scheduled__2021-05-03T17:05:00+00:00, externally triggered: False> successful
```

You can add and update your DAGs at anytime. These changes will be picked up by the scheduler on its next scheduling loop and should appear within 30 secs or so.

### As a Daemon

For users who are more comfortable with Airflow and wish to run it in the background, you can run your Airflow as a Daemon by running the following command:

```
1 airflow scheduler --daemon
```

To check if your Airflow scheduler is running, run the following command and look for a process with the name "airflow scheduler – DagFileProcessorManager":

```
1 ps aux | less
```

To kill your Airflow scheduler Daemon, you can run the following command:

```
1 pkill -f "airflow scheduler"
```

#### 6a. Use the Airflow webserver

The easiest way to visualize your DAG testing is by running your very own Airflow Web server, and with the help of JupyterHub's proxy - it's totally do-able!

⚠ Any user with access to JupyterHub can access your proxy server!

### How to setup the Airflow Web server on JupyterHub

Before running the Airflow Web server, make sure there exists a file named `airflow.cfg` in your `~/airflow` directory. If not, try re-running `airflow db init` to generate the file.

Once the file is generated, find the `enable_proxy_fix` property under the `[webserver]` section, and change its value to `True`.

```
1 [webserver]
2 ...
3 enable_proxy_fix = True
```

Furthermore, we will need to create a default user in order to access the UI. To do so, run the following command:

```
1 # This creates a user with Admin as everything (role, username, firstname, lastname, password)
2 airflow users create -r Admin -u <username> -e <email> -f <first_name> -l <last_name> -p <password>
```


Finally, run the Airflow webserver with the following command:

```
1 airflow webserver
```

After 15-30 seconds or so, you should be able to access your Airflow UI using the custom URL below:

```
1 https://jupyterhub.hogwarts.<environment>.azure.chimera.cyber.gc.ca/user/<last_name>_<first_name>_<initial>/prox
```

(make sure to change the environment (UDEV, U, PB) and the names accordingly)

 If you encounter a "limit request headers fields size" error, add `GUNICORN_CMD_ARGS='--limit-request-line 20000 --limit-request-field_size 20000'` before your `airflow webserver` command to fix it.

#### 6b. Use the Airflow CLI

For users who prefer a CLI based interface over a typical Web UI interface, Airflow provides its very own CLI solution to meet those needs.

 This solution is not available for the official Hogwarts Airflow platform. It is only a development tool.

Here are some interesting commands to get you started:

##### List your DAGs

```
1 airflow dags list
```

##### List DAG runs

```
1 airflow dags list-runs
2 airflow dags list-runs -d <dag_id>
```

##### Trigger your DAG (manual run)

```
1 airflow dags trigger <dag_id>
```

##### Print cheat-sheet (very helpful)

```
1 airflow cheat-sheet
```

## Deploying your DAG

With the arrival of Spellbook, creating and updating your Airflow DAG has never been easier. In fact, before your DAG is saved to Airflow, Spellbook will connect to Airflow and validate your DAG - preventing any invalid DAGs from being sent to the platform. This avoids all headaches caused by a simple typo to an invalid dependency import!

For more information on how to get started with Spellbook, click on the following link: [Spellbook](#).

## Templates and examples

To quick start your Airflow journey, the Hogwarts team offers a repository containing DAG templates that can help you start your next DAG.

[Hogwarts DAG templates](#)

## Internally developed Operators (Daggers)

The Hogwarts team offers *Daggers*, a set of Task operators to help with specific use-cases:

- **Git Operators:** Operators will pull a Github/Gitlab repository before running their specific code type (e.g. Python, Bash, Notebook)
- **ContainerOperator:** Hogwarts' version of KubernetesPodOperator. Abstracts specifics of the cluster for easier use.

For more information, visit the [Daggers repository](#).

## Feedback

We are always looking to improve our documentation. If you encountered any obstacles while going through this guide, feel free to reach out to us through our [Microsoft teams support channel](#).

## More information & links

### Official docs

- [Apache Airflow concepts](#)
- [Airflow macros](#)
- [Airflow operators](#)

### Internal docs

A few guides and articles have been written by the Hogwarts team to help you with your scheduling.

- [📖 Common Airflow mistakes](#)
- [📖 Rerunning a failed job](#)

### Great articles/resources

- [Towardsdatascience.com - Airflow scheduling 101](#)
- [Data pipelines with Apache Airflow \(book\)](#)