# Airflow operators

INTRODUCTION TO APACHE AIRFLOW IN PYTHON

**Mike Metzger**
Data Engineer

## Operators

- Represent a single task in a workflow.
- Run independently (usually).
- Generally do not share information.
- Various operators to perform different tasks.

```
# New way, Airflow 2.x+
EmptyOperator(task_id='example')

# Old way, Airflow <2.0
EmptyOperator(task_id='example', dag=dag_name)
```

## BashOperator

```
BashOperator(
    task_id='bash_example',
    bash_command='echo "Example!"',
    # Next line only for Airflow before v
    dag=dag
)
```

```
BashOperator(
    task_id='bash_script_example',
    bash_command='runcleanup.sh',
)
```

- Executes a given Bash command or script.
- Runs the command in a temporary directory.
- Can specify environment variables for the command.

## BashOperator examples

```python
from airflow.operators.bash import BashOperator
example_task = BashOperator(task_id='bash_ex',
                            bash_command='echo 1',
                           )
```

```python
bash_task = BashOperator(task_id='clean_addresses',
  bash_command='cat addresses.txt | awk "NF==10" > cleaned.txt',
)
```

## Operator gotchas

- Not guaranteed to run in the same location / environment.
- May require extensive use of Environment variables.
- Can be difficult to run tasks with elevated privileges.

## Let's practice!

INTRODUCTION TO APACHE AIRFLOW IN PYTHON

# Airflow tasks

INTRODUCTION TO APACHE AIRFLOW IN PYTHON

**Mike Metzger**
Data Engineer

# Tasks

*Tasks* are:

- Instances of operators
- Usually assigned to a variable in Python

```
example_task = BashOperator(task_id='bash_example',
                            bash_command='echo "Example!"')
```

- Referred to by the task_id within the Airflow tools

## Task dependencies

- Define a given order of task completion
- Are not required for a given workflow, but usually present in most
- Are referred to as *upstream* or *downstream* tasks
- In Airflow 1.8 and later, are defined using the *bitshift* operators
  - >>, or the upstream operator
  - <<, or the downstream operator

## Upstream vs Downstream

*Upstream* means **before**

*Downstream* means **after**

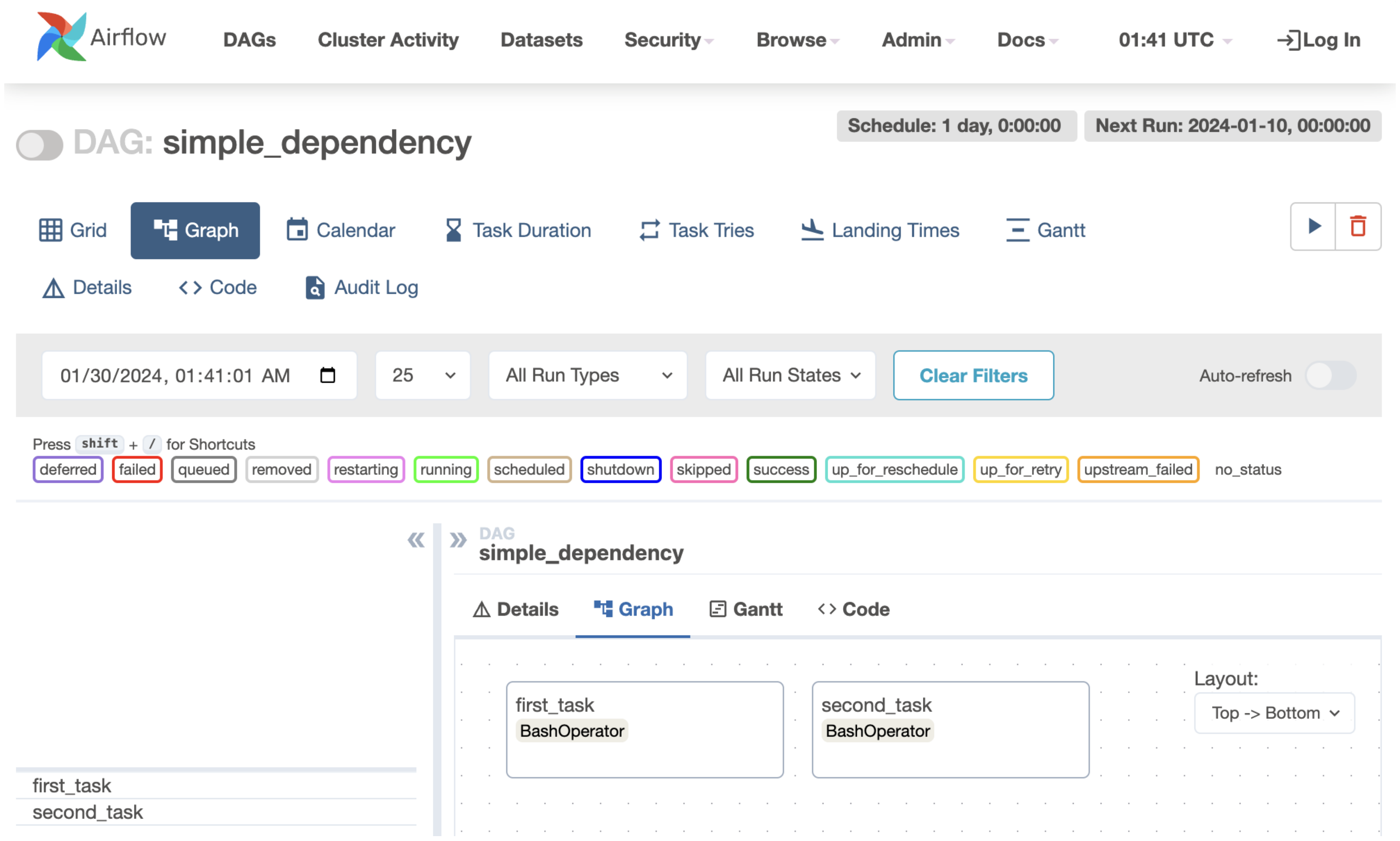
## Simple task dependency

```
# Define the tasks
task1 = BashOperator(task_id='first_task',
                     bash_command='echo 1'
                    )

task2 = BashOperator(task_id='second_task',
                     bash_command='echo 2'
                    )

# Set first_task to run before second_task
task1 >> task2   # or task2 << task1
```

# Task dependencies in the Airflow UI



DAG: simple_dependency

Schedule: 1 day, 0:00:00 | Next Run: 2024-01-10, 00:00:00

01/30/2024, 01:41:01 AM | 25 | All Run Types | All Run States | Clear Filters | Auto-refresh

Press shift + / for Shortcuts

deferred failed queued removed restarting running scheduled shutdown skipped success up_for_reschedule up_for_retry upstream_failed no_status

DAG
simple_dependency

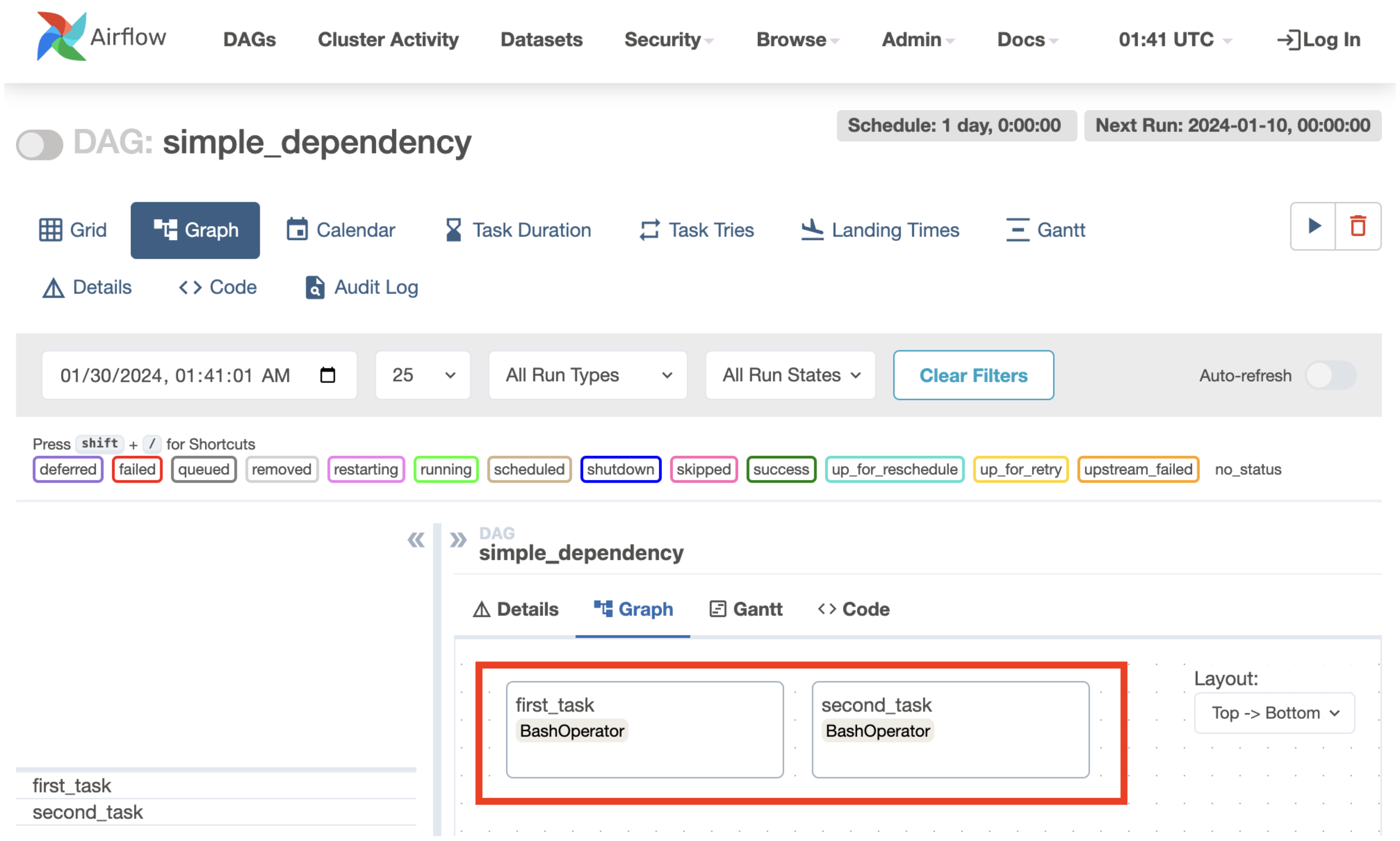
first_task
BashOperator

second_task
BashOperator

Layout: Top -> Bottom

first_task
second_task

# Task dependencies in the Airflow UI



Airflow DAGs Cluster Activity Datasets Security Browse Admin Docs 01:41 UTC Log In

DAG: simple_dependency

Schedule: 1 day, 0:00:00 Next Run: 2024-01-10, 00:00:00

Grid Graph Calendar Task Duration Task Tries Landing Times Gantt

Details Code Audit Log

01/30/2024, 01:41:01 AM | 25 | All Run Types | All Run States | Clear Filters | Auto-refresh

Press shift + / for Shortcuts

deferred failed queued removed restarting running scheduled shutdown skipped success up_for_reschedule up_for_retry upstream_failed no_status

DAG simple_dependency

Details Graph Gantt Code

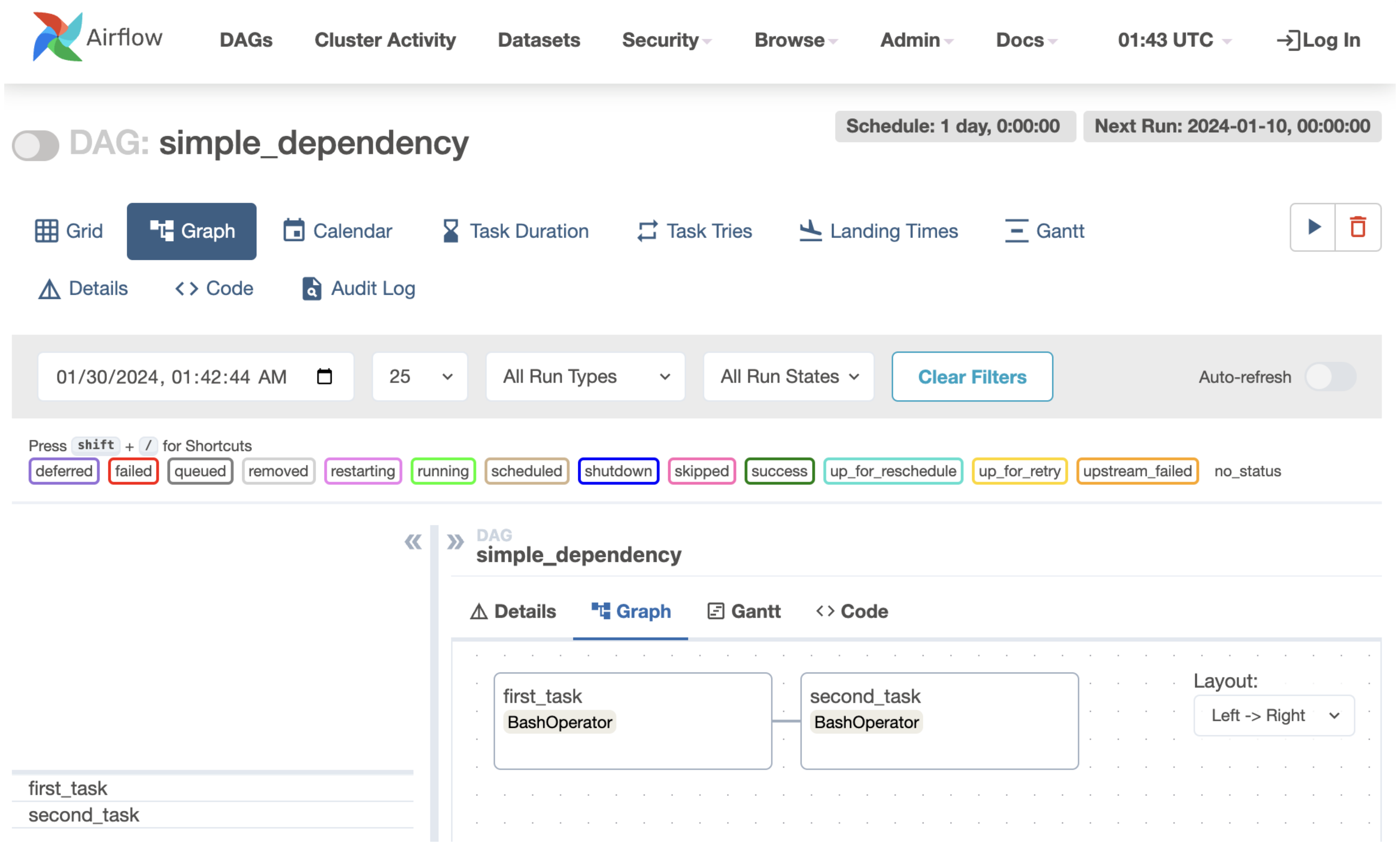
first_task BashOperator

second_task BashOperator

Layout: Top -> Bottom

first_task
second_task

# Task dependencies in the Airflow UI



DAG: simple_dependency

Schedule: 1 day, 0:00:00
Next Run: 2024-01-10, 00:00:00

01/30/2024, 01:42:44 AM
25
All Run Types
All Run States

DAG
simple_dependency

first_task
BashOperator

second_task
BashOperator

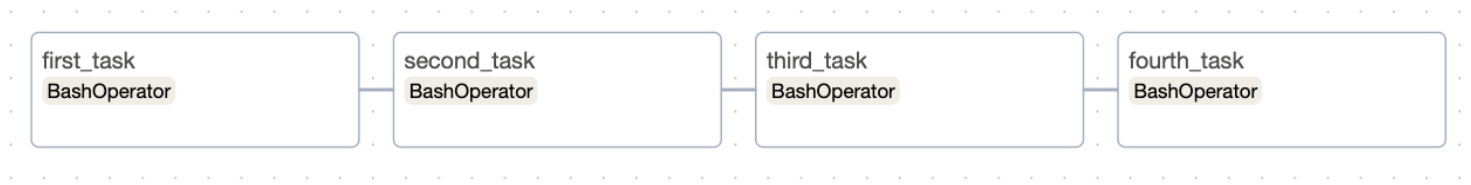
Layout: Left -> Right

first_task
second_task

## Multiple dependencies

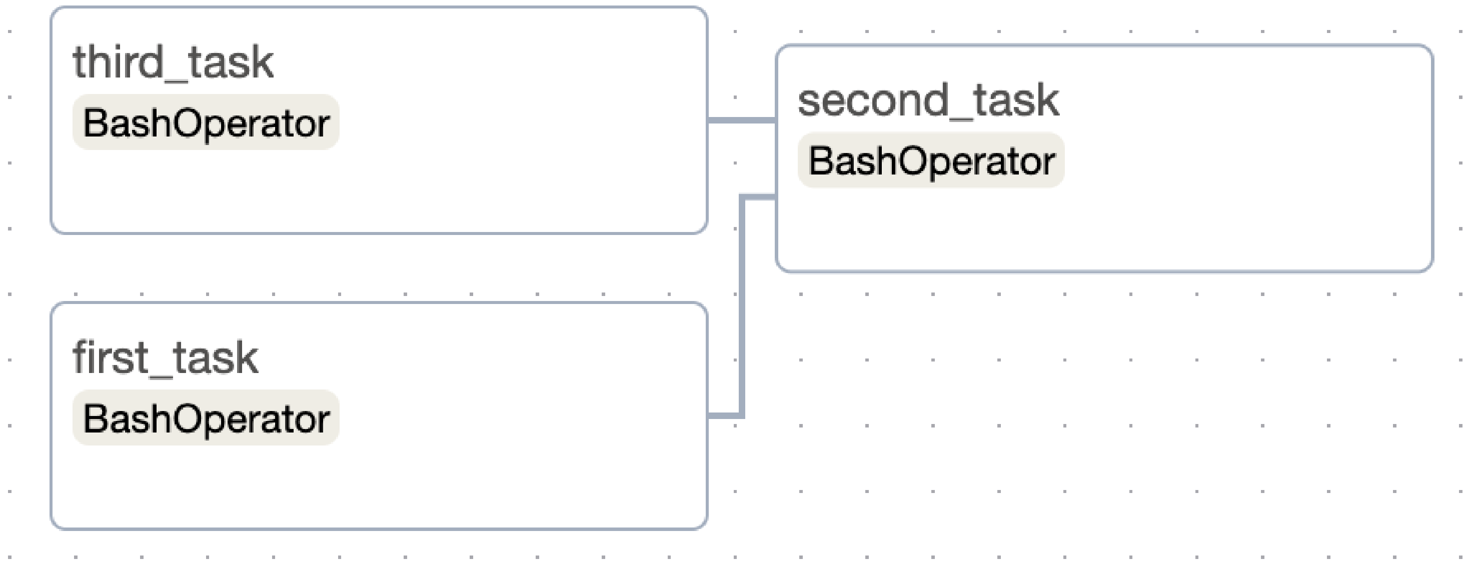
Chained dependencies:

```
task1 >> task2 >> task3 >> task4
```



Mixed dependencies:

```
task1 >> task2 << task3
```



or:

```
task1 >> task2
task3 >> task2
```

## Let's practice!

INTRODUCTION TO APACHE AIRFLOW IN PYTHON

# Additional operators

INTRODUCTION TO APACHE AIRFLOW IN PYTHON

Mike Metzger
Data Engineer

## PythonOperator

- Executes a Python function / callable
- Operates similarly to the BashOperator, with more options
- Can pass in arguments to the Python code

```python
from airflow.operators.python import PythonOperator
def printme():
    print("This goes in the logs!")
python_task = PythonOperator(
    task_id='simple_print',
    python_callable=printme
)
```

## Arguments

- Supports arguments to tasks
  - Positional
  - Keyword
- Use the `op_kwargs` dictionary

## op_kwargs example

```
def sleep(length_of_time):
  time.sleep(length_of_time)

sleep_task = PythonOperator(
    task_id='sleep',
    python_callable=sleep,
    op_kwargs={'length_of_time': 5}
)
```

## EmailOperator

- Found in the `airflow.operators` library
- Sends an email
- Can contain typical components
  - HTML content
  - Attachments
- Does require the Airflow system to be configured with email server details

## EmailOperator example

```python
from airflow.operators.email import EmailOperator

email_task = EmailOperator(
    task_id='email_sales_report',
    to='sales_manager@example.com',
    subject='Automated Sales Report',
    html_content='Attached is the latest sales report',
    files='latest_sales.xlsx'
)
```

## Let's practice!

INTRODUCTION TO APACHE AIRFLOW IN PYTHON

# Airflow scheduling

INTRODUCTION TO APACHE AIRFLOW IN PYTHON

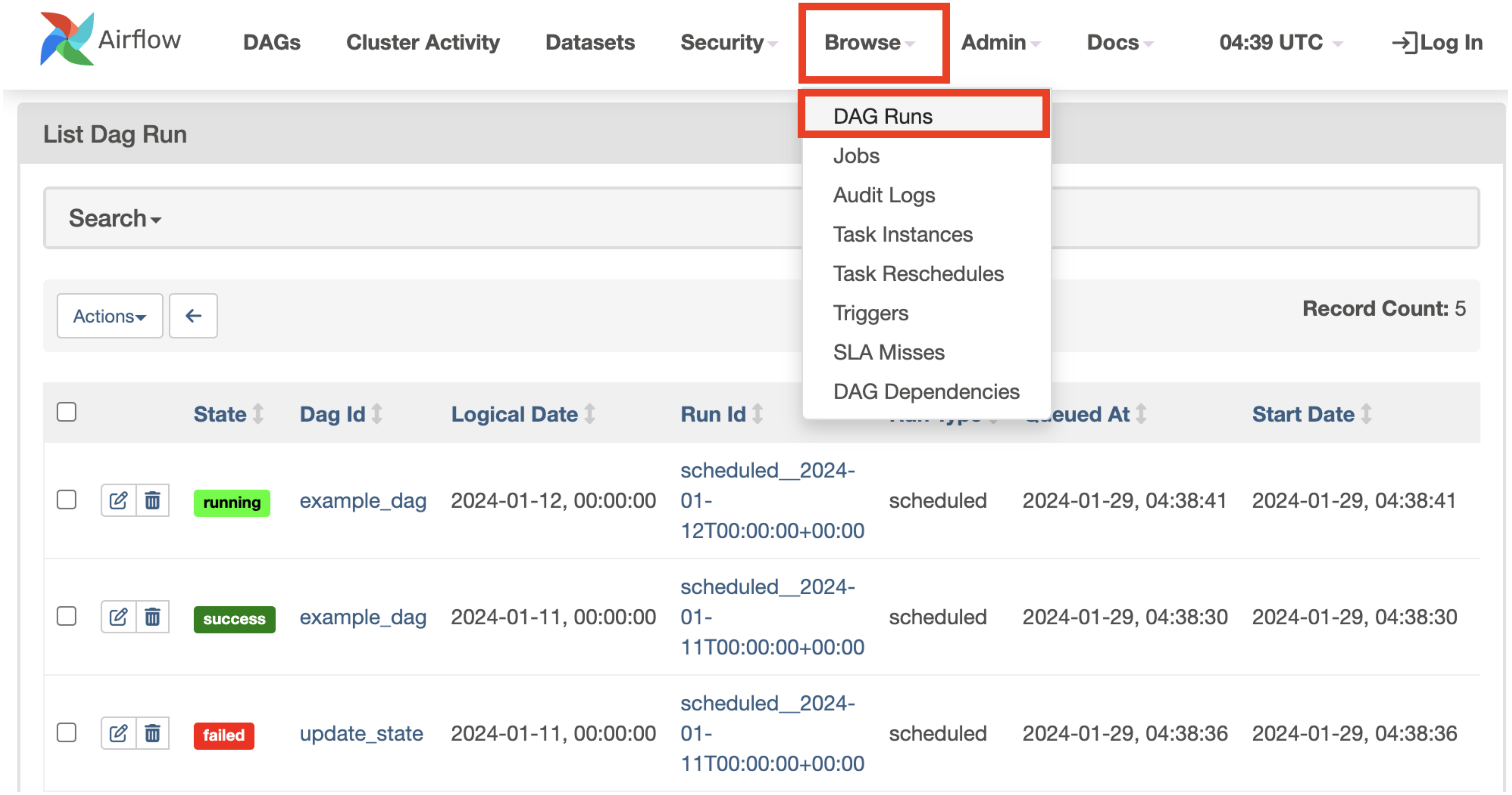
**Mike Metzger**
Data Engineer

## DAG Runs

- A specific instance of a workflow at a point in time
- Can be run manually or via `schedule_interval`
- Maintain state for each workflow and the tasks within
  - `running`
  - `failed`
  - `success`

[1] https://airflow.apache.org/docs/stable/scheduler.html

# DAG Runs view

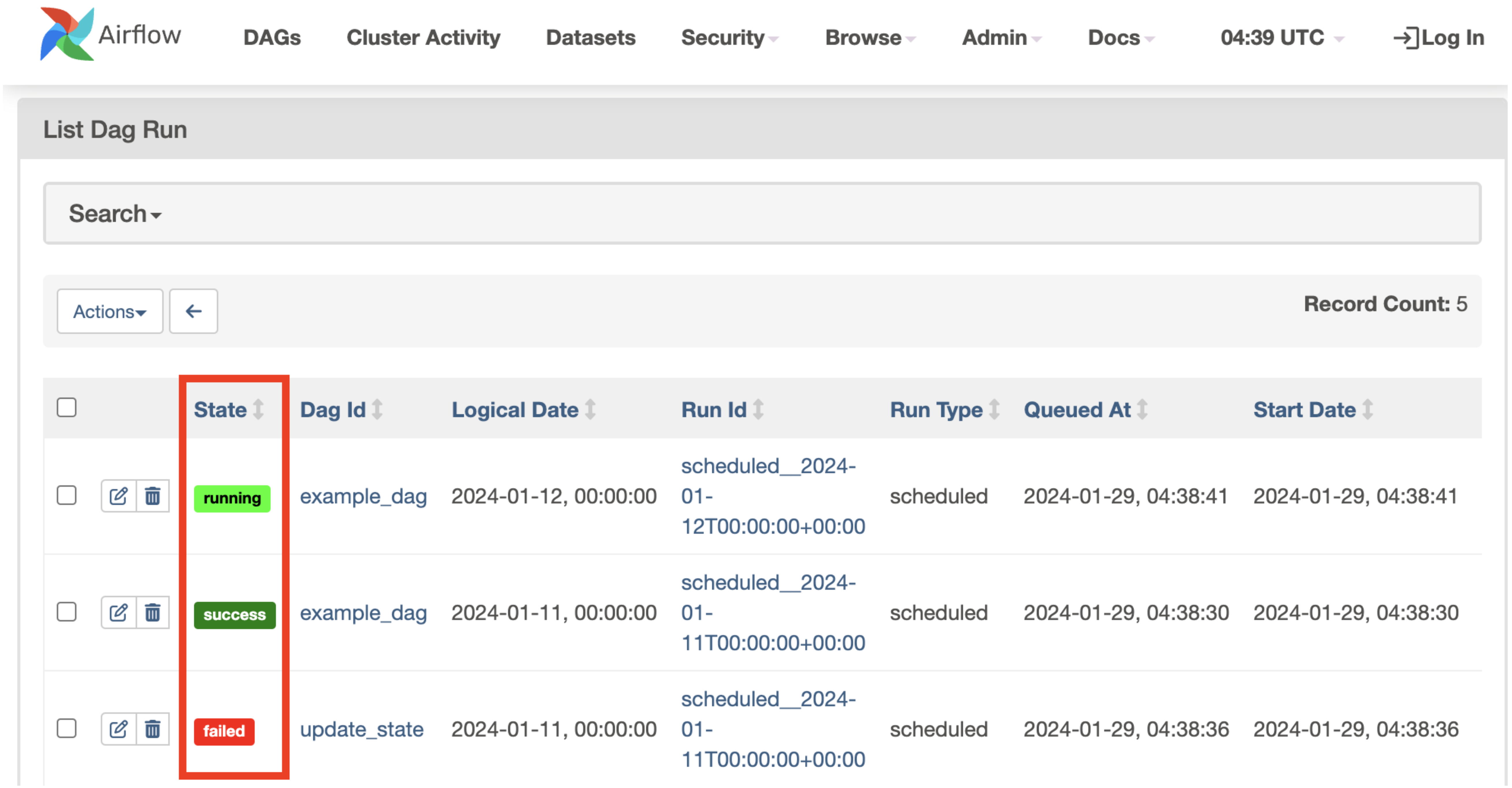


List Dag Run

**Record Count:** 5

| | State | Dag Id | Logical Date | Run Id | Run Type | Queued At | Start Date |
|---|---|---|---|---|---|---|---|
| | running | example_dag | 2024-01-12, 00:00:00 | scheduled__2024-01-12T00:00:00+00:00 | scheduled | 2024-01-29, 04:38:41 | 2024-01-29, 04:38:41 |
| | success | example_dag | 2024-01-11, 00:00:00 | scheduled__2024-01-11T00:00:00+00:00 | scheduled | 2024-01-29, 04:38:30 | 2024-01-29, 04:38:30 |
| | failed | update_state | 2024-01-11, 00:00:00 | scheduled__2024-01-11T00:00:00+00:00 | scheduled | 2024-01-29, 04:38:36 | 2024-01-29, 04:38:36 |

# DAG Runs state



List Dag Run

Record Count: 5

| State | Dag Id | Logical Date | Run Id | Run Type | Queued At | Start Date |
|---|---|---|---|---|---|---|
| running | example_dag | 2024-01-12, 00:00:00 | scheduled__2024-01-12T00:00:00+00:00 | scheduled | 2024-01-29, 04:38:41 | 2024-01-29, 04:38:41 |
| success | example_dag | 2024-01-11, 00:00:00 | scheduled__2024-01-11T00:00:00+00:00 | scheduled | 2024-01-29, 04:38:30 | 2024-01-29, 04:38:30 |
| failed | update_state | 2024-01-11, 00:00:00 | scheduled__2024-01-11T00:00:00+00:00 | scheduled | 2024-01-29, 04:38:36 | 2024-01-29, 04:38:36 |

## Schedule details

When scheduling a DAG, there are several attributes of note:

- `start_date` - The date / time to initially schedule the DAG run
- `end_date` - Optional attribute for when to stop running new DAG instances
- `max_tries` - Optional attribute for how many attempts to make
- `schedule_interval` - How often to run

## Schedule interval

`schedule_interval` represents:

- How often to schedule the DAG
- Between the `start_date` and `end_date`
- Can be defined via `cron` style syntax or via built-in presets.

## cron syntax

```
# ┌───────────── 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)
# │ │ │ │ │
# │ │ │ │ │
# * * * * * command to execute
```

- Is pulled from the Unix cron format
- Consists of 5 fields separated by a space
- An asterisk `*` represents running for every interval (ie, every minute, every day, etc)
- Can be comma separated values in fields for a list of values

## cron examples

```
0 12 * * *                  # Run daily at noon
```

```
* * 25 2 *                  # Run once per minute on February 25
```

```
0,15,30,45 * * * *          # Run every 15 minutes
```

# Airflow scheduler presets

| Preset: | cron equivalent: |
|---|---|
| @hourly | `0 * * * *` |
| @daily | `0 0 * * *` |
| @weekly | `0 0 * * 0` |
| @monthly | `0 0 1 * *` |
| @yearly | `0 0 1 1 *` |

[1] https://airflow.apache.org/docs/stable/scheduler.html

## Special presets

Airflow has two special `schedule_interval` presets:

- `None` - Don't schedule ever, used for manually triggered DAGs
- `@once` - Schedule only once

## schedule_interval issues

When scheduling a DAG, Airflow will:

- Use the `start_date` as the earliest possible value
- Schedule the task at `start_date` + `schedule_interval`

```
'start_date': datetime(2020, 2, 25),
'schedule_interval': @daily
```

This means the earliest starting time to run the DAG is on February 26th, 2020

## Let's practice!