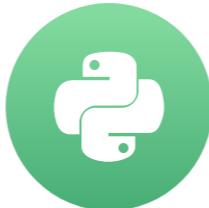


Introduction to Airflow

INTRODUCTION TO AIRFLOW IN PYTHON



Mike Metzger
Data Engineer

What is data engineering?

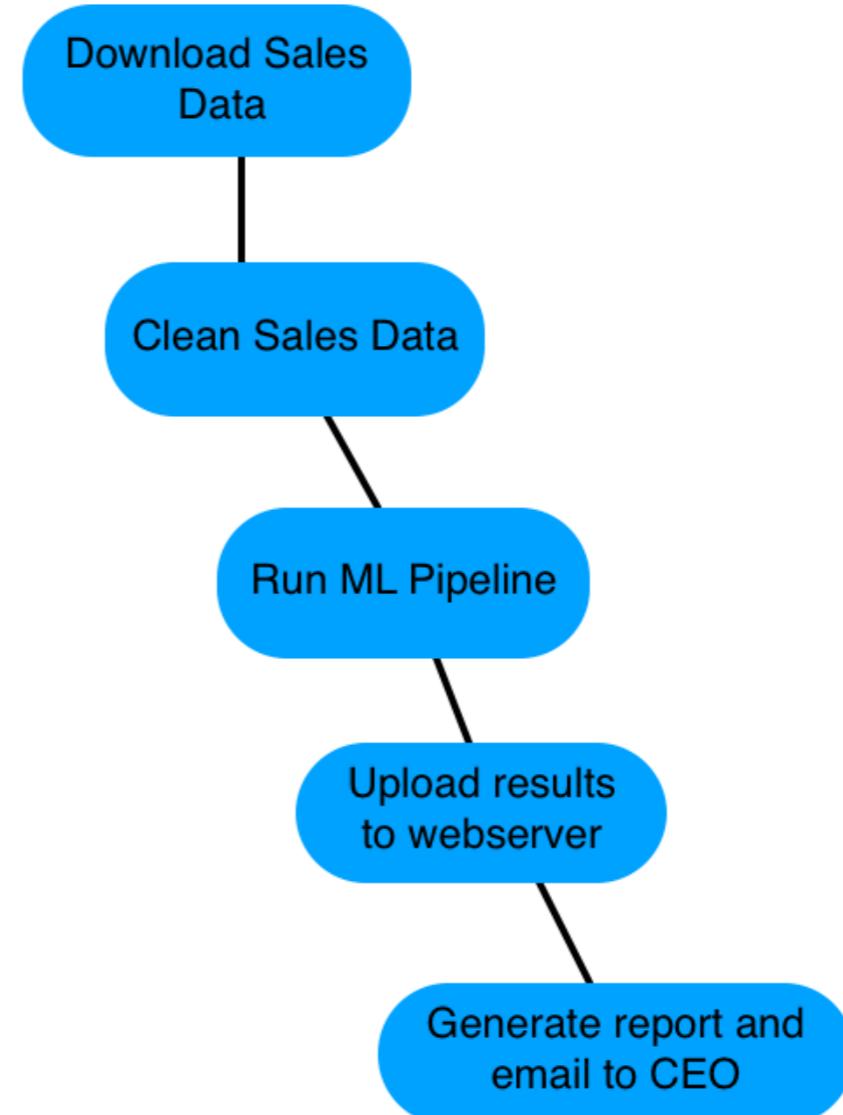
Data engineering is:

- Taking any action involving data and turning it into a reliable, repeatable, and maintainable process.

What is a workflow?

A *workflow* is:

- A set of steps to accomplish a given data engineering task
 - Such as: downloading files, copying data, filtering information, writing to a database, etc
- Of varying levels of complexity
- A term with various meaning depending on context



What is Airflow?

Airflow is a platform to program workflows,
including:

- Creation
- Scheduling
- Monitoring



Apache
Airflow

Airflow continued...

- Can implement programs from any language, but workflows are written in Python
- Implements workflows as DAGs: Directed Acyclic Graphs
- Accessed via code, command-line, or via web interface



Apache
Airflow

¹ <https://airflow.apache.org/docs/stable/>

Other workflow tools

Other tools:

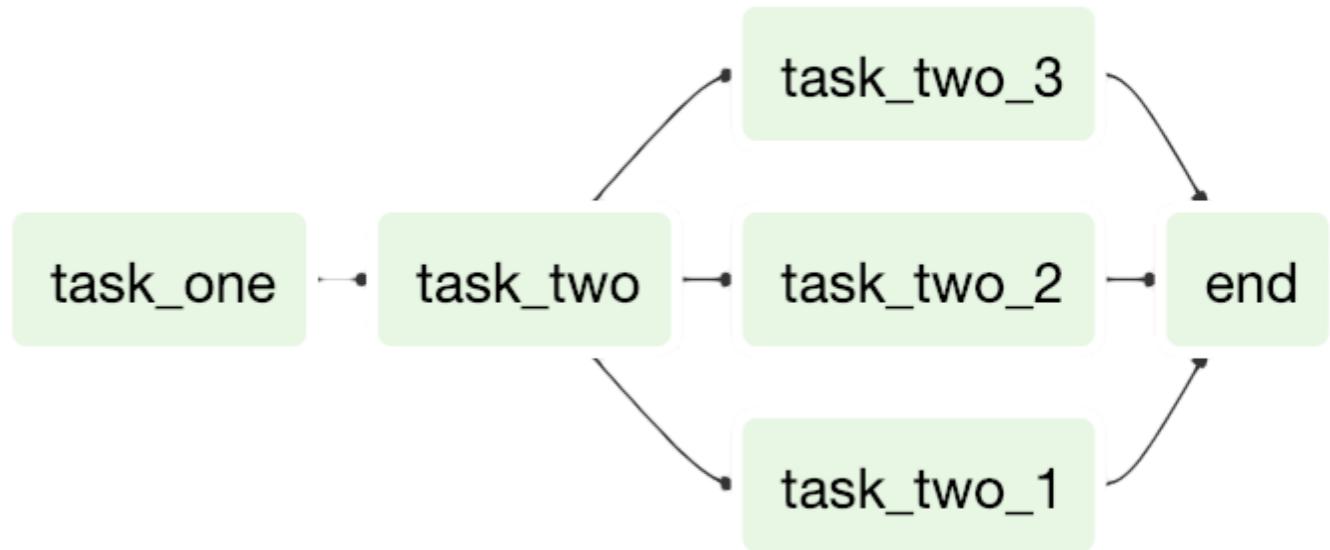
- Luigi
- SSIS
- Bash scripting



Quick introduction to DAGs

A *DAG* stands for *Directed Acyclic Graph*

- In Airflow, this represents the set of tasks that make up your workflow.
- Consists of the tasks and the dependencies between tasks.
- Created with various details about the DAG, including the name, start date, owner, etc.
- Further depth in the next lesson.



DAG code example

Simple DAG definition:

```
etl_dag = DAG(  
    dag_id='etl_pipeline',  
    default_args={"start_date": "2020-01-08"}  
)
```

Running a workflow in Airflow

Running a simple Airflow task

```
airflow run <dag_id> <task_id> <start_date>
```

Using a DAG named *example-etl*, a task named *download-file* and a start date of 2020-01-10:

```
airflow run example-etl download-file 2020-01-10
```

Let's practice!

INTRODUCTION TO AIRFLOW IN PYTHON

Airflow DAGs

INTRODUCTION TO AIRFLOW IN PYTHON

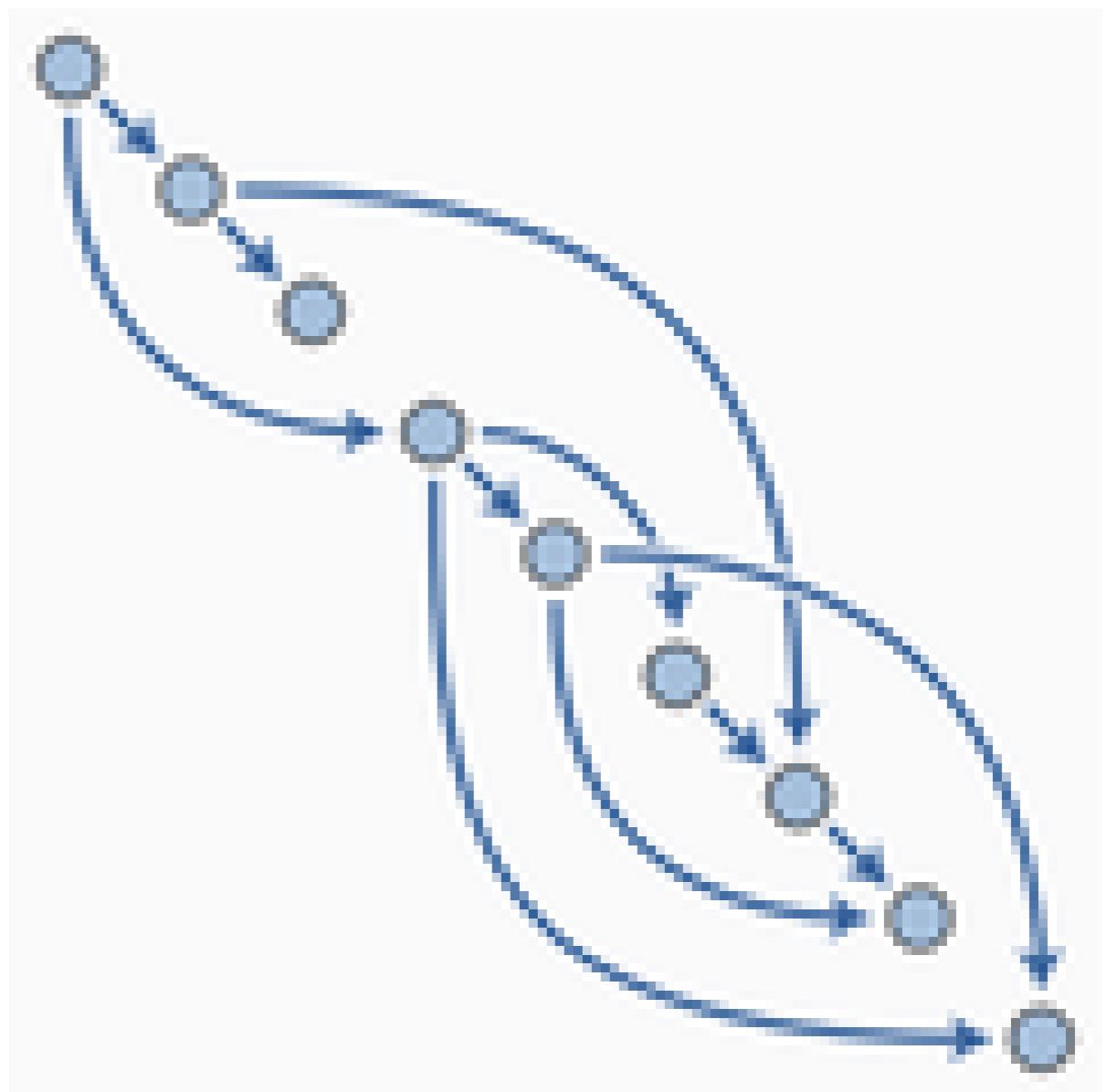


Mike Metzger
Data Engineer

What is a DAG?

DAG, or *Directed Acyclic Graph*:

- *Directed*, there is an inherent flow representing dependencies between components.
- *Acyclic*, does not loop / cycle / repeat.
- *Graph*, the actual set of components.
- Seen in Airflow, Apache Spark, Luigi



¹ https://en.m.wikipedia.org/wiki/Directed_acyclic_graph

DAG in Airflow

Within Airflow, DAGs:

- Are written in Python (but can use components written in other languages).
- Are made up of components (typically *tasks*) to be executed, such as operators, sensors, etc.
- Contain dependencies defined explicitly or implicitly.
 - ie, Copy the file to the server before trying to import it to the database service.

Define a DAG

Example DAG:

```
from airflow.models import DAG

from datetime import datetime
default_arguments = {
    'owner': 'jdoe',
    'email': 'jdoe@datacamp.com',
    'start_date': datetime(2020, 1, 20)
}

etl_dag = DAG('etl_workflow', default_args=default_arguments)
```

DAGs on the command line

Using `airflow` :

- The `airflow` command line program contains many subcommands.
- `airflow -h` for descriptions.
- Many are related to DAGs.
- `airflow list_dags` to show all recognized DAGs.

Command line vs Python

Use the command line tool to:

- Start Airflow processes
- Manually run DAGs / Tasks
- Get logging information from Airflow

Use Python to:

- Create a DAG
- Edit the individual properties of a DAG

Let's practice!

INTRODUCTION TO AIRFLOW IN PYTHON

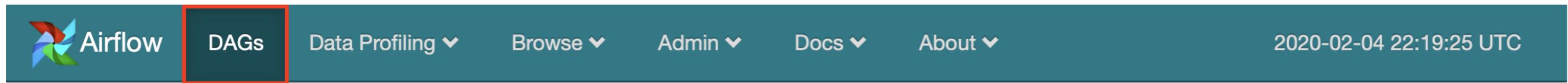
Airflow web interface

INTRODUCTION TO AIRFLOW IN PYTHON



Mike Metzger
Data Engineer

DAGs view



DAGs

Search:

		DAG	Schedule	Owner	Recent Tasks	Last Run	DAG Runs	Links
		example_dag	1 day, 0:00:00	airflow	(10 total)	(3 total)	(3 total)	
		update_state	1 day, 0:00:00	airflow	(10 total)	(3 total)	(3 total)	

Showing 1 to 2 of 2 entries

« < 1 > »

DAGs view DAGs

 Airflow DAGs Data Profiling ▾ Browse ▾ Admin ▾ Docs ▾ About ▾ 2020-02-04 22:19:25 UTC

DAGs

Search:

	 DAG	Schedule	Owner	Recent Tasks 	 Last Run	 DAG Runs 	Links
 	example_dag	1 day, 0:00:00	airflow	                     <img alt			

DAGs view schedule

Airflow DAGs Data Profiling ▾ Browse ▾ Admin ▾ Docs ▾ About ▾ 2020-02-04 22:19:25 UTC

DAGs

Search:

i	DAG	Schedule	Owner	Recent Tasks i	Last Run i	DAG Runs i	Links
	Off example_dag	1 day, 0:00:00	airflow				
	Off update_state	1 day, 0:00:00	airflow				

Showing 1 to 2 of 2 entries

« < 1 > »

DAGs view owner

Airflow DAGs Data Profiling ▾ Browse ▾ Admin ▾ Docs ▾ About ▾ 2020-02-04 22:19:25 UTC

DAGs

Search:

i	DAG	Schedule	Owner	Recent Tasks i	Last Run i	DAG Runs i	Links
	example_dag	1 day, 0:00:00	airflow				
	update_state	1 day, 0:00:00	airflow				

Showing 1 to 2 of 2 entries

« < 1 > »

DAGs view recent tasks

Airflow DAGs Data Profiling ▾ Browse ▾ Admin ▾ Docs ▾ About ▾ 2020-02-04 22:19:25 UTC

DAGs

Search:

i	DAG	Schedule	Owner	Recent Tasks i	Last Run i	DAG Runs i	Links
Off	example_dag	1 day, 0:00:00	airflow	○ ○ ○ ○ ○ ○ ○ ○	○	○ ○ ○	
Off	update_state	1 day, 0:00:00	airflow	○ ○ ○ ○ ○ ○ ○ ○	○	○ ○ ○	

Showing 1 to 2 of 2 entries

« < 1 > »

DAGs view last run

Airflow DAGs Data Profiling ▾ Browse ▾ Admin ▾ Docs ▾ About ▾ 2020-02-04 22:19:25 UTC

DAGs

Search:

i	DAG	Schedule	Owner	Recent Tasks i	Last Run i	DAG Runs i	Links
	Off example_dag	1 day, 0:00:00	airflow				
	Off update_state	1 day, 0:00:00	airflow				

Showing 1 to 2 of 2 entries

« < 1 > »

DAGs view last three

Airflow DAGs Data Profiling ▾ Browse ▾ Admin ▾ Docs ▾ About ▾ 2020-02-04 22:19:25 UTC

DAGs

Search:

	i DAG	Schedule	Owner	Recent Tasks i	Last Run i	DAG Runs i	Links
	Off example_dag 1 day, 0:00:00	airflow					
	Off update_state 1 day, 0:00:00	airflow					

Showing 1 to 2 of 2 entries

« < 1 > »

DAGs view links

Airflow DAGs Data Profiling ▾ Browse ▾ Admin ▾ Docs ▾ About ▾ 2020-02-04 22:19:25 UTC

DAGs

Search:

	i DAG	Schedule	Owner	Recent Tasks i	i Last Run	DAG Runs i	Links
	Off example_dag 1 day, 0:00:00 airflow						
	Off update_state 1 day, 0:00:00 airflow						

Showing 1 to 2 of 2 entries

« < 1 > »

DAGs view example_dag

Airflow DAGs Data Profiling ▾ Browse ▾ Admin ▾ Docs ▾ About ▾ 2020-02-04 22:19:25 UTC

DAGs

Search:

	i DAG	Schedule	Owner	Recent Tasks i	i Last Run	DAG Runs i	Links
	Off example_dag	1 day, 0:00:00	airflow				
	Off update_state	1 day, 0:00:00	airflow				

Showing 1 to 2 of 2 entries

« < 1 > »

DAG detail view

Airflow DAGs Data Profiling ▾ Browse ▾ Admin ▾ Docs ▾ About ▾ 2020-01-22 14:24:46 UTC

Off DAG: example_dag schedule: 1 day, 0:00:00

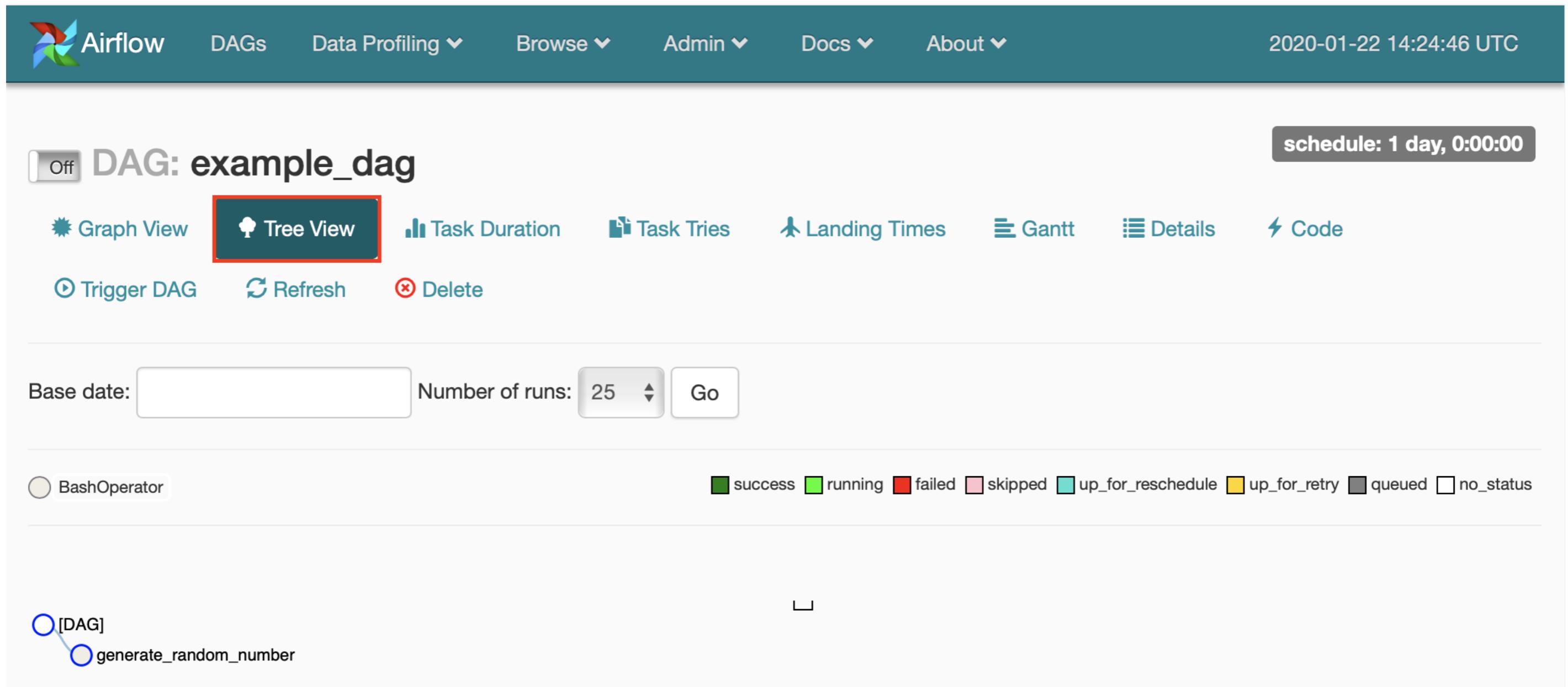
Graph View Tree View Task Duration Task Tries Landing Times Gantt Details Code

Trigger DAG Refresh Delete

Base date: Number of runs: 25 Go

BashOperator success running failed skipped up_for_reschedule up_for_retry queued no_status

[DAG] generate_random_number



DAG graph view

Airflow DAGs Data Profiling ▾ Browse ▾ Admin ▾ Docs ▾ About ▾ 2020-01-22 14:37:28 UTC

Off DAG: example_dag schedule: 1 day, 0:00:00

Graph View Tree View Task Duration Task Tries Landing Times Gantt Details Code

Trigger DAG Refresh Delete

None Base date: 2020-01-22 14:34:23 Number of runs: 25 Run: Layout: Left->Right Go Search for...

BashOperator success running failed skipped up_for_reschedule up_for_retry queued no_status

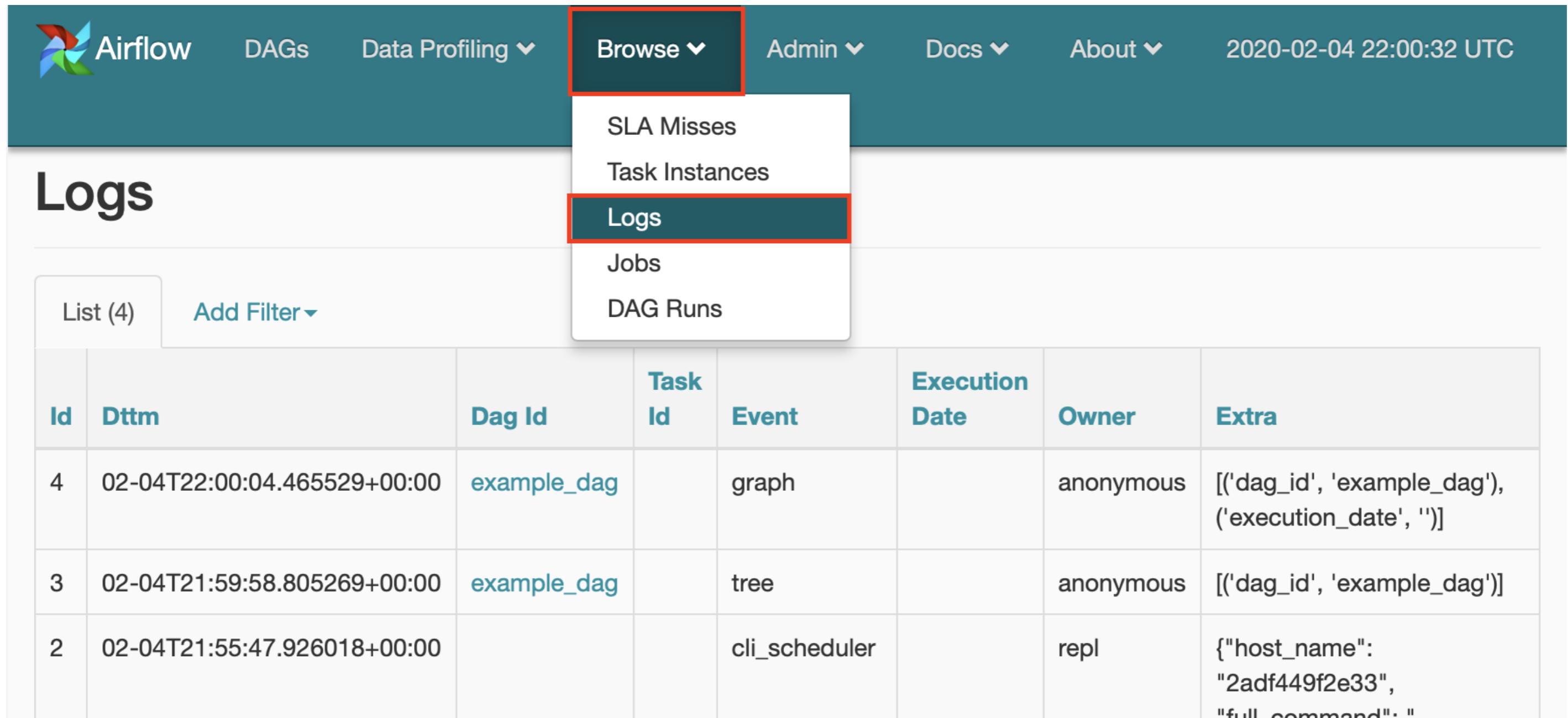
generate_random_number

DAG code view

The screenshot shows the Airflow web interface for a DAG named "example_dag". The top navigation bar includes links for Airflow, DAGs, Data Profiling, Browse, Admin, Docs, and About, along with a timestamp of 2020-01-23 14:48:35 UTC. Below the navigation, there's a status indicator showing "Off" and a schedule time of "schedule: 1 day, 0:00:00". A toolbar below the title provides various visualization options: Graph View, Tree View, Task Duration, Task Tries, Landing Times, Gantt, Details, Trigger DAG, Refresh, and Delete. The "Code" button is highlighted with a red box. The main content area displays the Python code for the DAG:

```
1 from airflow.models import DAG
2 from airflow.operators.bash_operator import BashOperator
3
4 dag = DAG(
5     dag_id = 'example_dag',
6     default_args={"start_date": "2019-10-01"}
7 )
8
9 part1 = BashOperator(
10    task_id='generate_random_number',
11    bash_command='echo $RANDOM',
12    dag=dag
13 )
```

Logs



The screenshot shows the Airflow web interface with a dark teal header bar. The header contains the Airflow logo, navigation links for DAGs, Data Profiling, Browse (which is currently selected and highlighted with a red box), Admin, Docs, and About, along with a timestamp of 2020-02-04 22:00:32 UTC.

The main content area has a title "Logs". Below it is a table with the following columns: Id, Dttm, Dag Id, Task Id, Event, Execution Date, Owner, and Extra. The table displays three log entries:

Id	Dttm	Dag Id	Task Id	Event	Execution Date	Owner	Extra
4	02-04T22:00:04.465529+00:00	example_dag		graph		anonymous	[('dag_id', 'example_dag'), ('execution_date', '')]
3	02-04T21:59:58.805269+00:00	example_dag		tree		anonymous	[('dag_id', 'example_dag')]
2	02-04T21:55:47.926018+00:00			cli_scheduler		repl	{"host_name": "2adf449f2e33", "full_command": ""}

Web UI vs command line

In most cases:

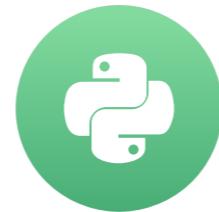
- Equally powerful depending on needs
- Web UI is easier
- Command line tool may be easier to access depending on settings

Let's practice!

INTRODUCTION TO AIRFLOW IN PYTHON

Airflow operators

INTRODUCTION TO 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.

```
DummyOperator(task_id='example', dag=dag)
```

BashOperator

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

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

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

BashOperator examples

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

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

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 AIRFLOW IN PYTHON

Airflow tasks

INTRODUCTION TO 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!"',  
                            dag=dag)
```

- 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',  
                     dag=example_dag)  
  
task2 = BashOperator(task_id='second_task',  
                     bash_command='echo 2',  
                     dag=example_dag)  
  
# Set first_task to run before second_task  
task1 >> task2 # or task2 << task1
```

Task dependencies in the Airflow UI

Airflow DAGs Data Profiling ▾ Browse ▾ Admin ▾ Docs ▾ About ▾ 2020-02-12 05:27:33 UTC

Off DAG: simple_dependency schedule: 1 day, 0:00:00

Graph View Tree View Task Duration Task Tries Landing Times Gantt Details Code Trigger DAG Refresh Delete

None Base date: 2020-02-12 05:26:17 Number of runs: 25 Run: Layout: Left->Right Go Search for...

BashOperator success running failed skipped up_for_reschedule up_for_retry queued no_status

first_task

second_task

↻

Task dependencies in the Airflow UI

Airflow DAGs Data Profiling ▾ Browse ▾ Admin ▾ Docs ▾ About ▾ 2020-02-12 05:27:33 UTC

Off DAG: simple_dependency schedule: 1 day, 0:00:00

Graph View Tree View Task Duration Task Tries Landing Times Gantt Details Code Trigger DAG Refresh Delete

None Base date: 2020-02-12 05:26:17 Number of runs: 25 Run: Layout: Left->Right Go Search for...

BashOperator success running failed skipped up_for_reschedule up_for_retry queued no_status

```
graph TD; first_task --> second_task
```

Task dependencies in the Airflow UI

Airflow DAGs Data Profiling ▾ Browse ▾ Admin ▾ Docs ▾ About ▾ 2020-02-12 05:38:17 UTC

Off DAG: simple_dependency schedule: 1 day, 0:00:00

Graph View Tree View Task Duration Task Tries Landing Times Gantt Details Code Trigger DAG Refresh Delete

None Base date: 2020-02-12 05:37:25 Number of runs: 25 Run: Layout: Left->Right Go Search for...
BashOperator success running failed skipped up_for_reschedule up_for_retry queued no_status

first_task → second_task

↻

```
graph LR; first_task --> second_task
```

Multiple dependencies

Chained dependencies:

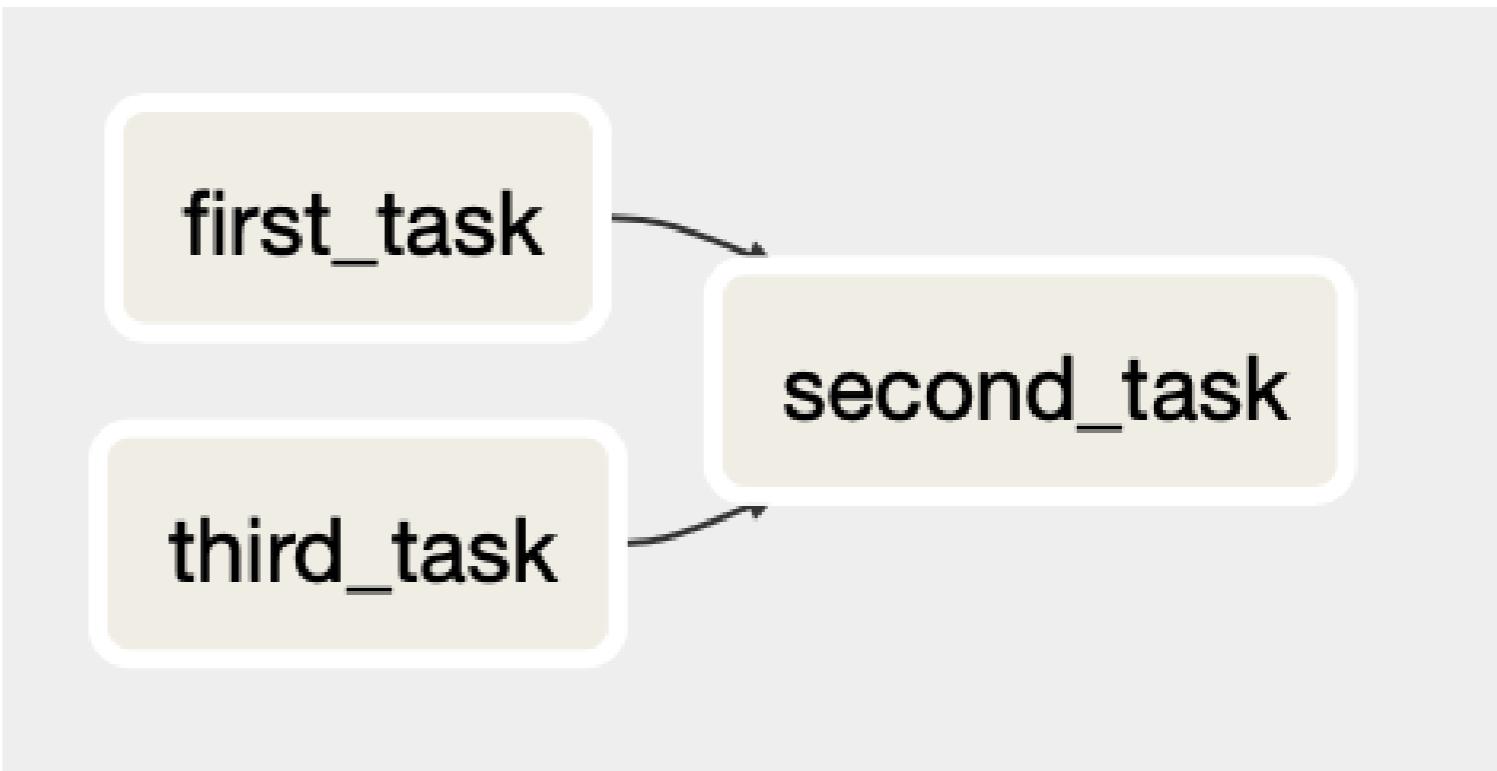
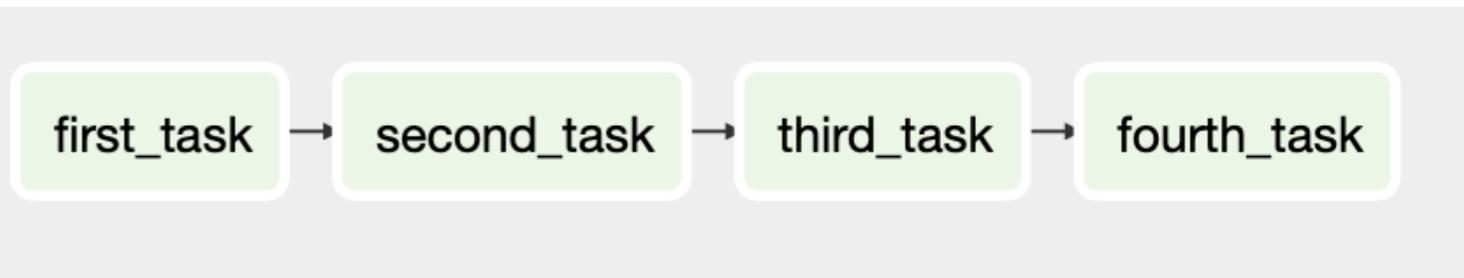
```
task1 >> task2 >> task3 >> task4
```

Mixed dependencies:

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

or:

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

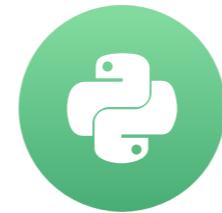


Let's practice!

INTRODUCTION TO AIRFLOW IN PYTHON

Additional operators

INTRODUCTION TO 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

```
from airflow.operators.python_operator import PythonOperator

def printme():
    print("This goes in the logs!")

python_task = PythonOperator(
    task_id='simple_print',
    python_callable=printme,
    dag=example_dag
)
```

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}  
    dag=example_dag  
)
```

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

```
from airflow.operators.email_operator 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',
    dag=example_dag
)
```

Let's practice!

INTRODUCTION TO AIRFLOW IN PYTHON

Airflow scheduling

INTRODUCTION TO 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



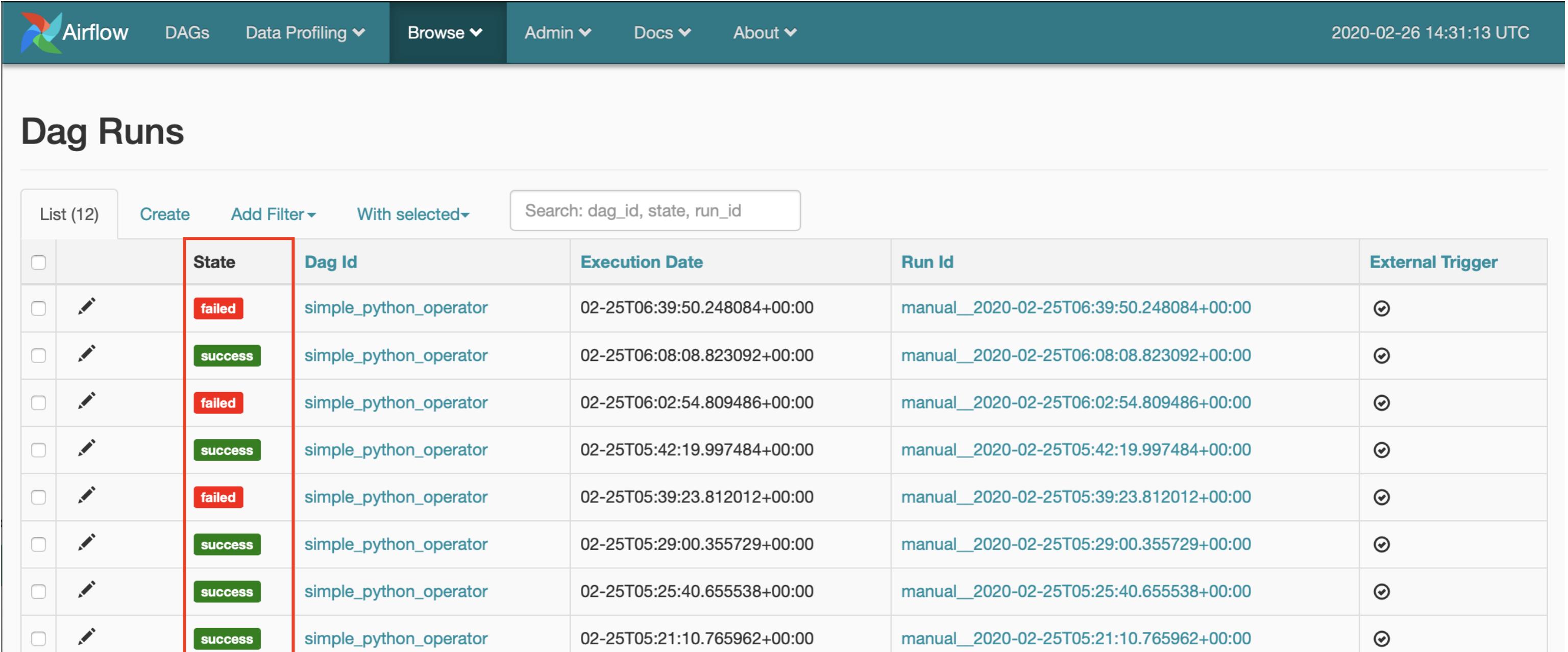
Airflow DAGs Data Profiling ▾ Browse ▾ Admin ▾ Docs ▾ About ▾ 2020-02-26 14:31:13 UTC

Dag Runs

List (12) Create Add Filter ▾ With selected ▾ Search: dag_id, state, run_id

<input type="checkbox"/>	State	Dag Id	Execution Date	Run Id	External Trigger
<input type="checkbox"/>	failed	simple_python_operator	02-25T06:39:50.248084+00:00	manual_2020-02-25T06:39:50.248084+00:00	<input checked="" type="checkbox"/>
<input type="checkbox"/>	success	simple_python_operator	02-25T06:08:08.823092+00:00	manual_2020-02-25T06:08:08.823092+00:00	<input checked="" type="checkbox"/>
<input type="checkbox"/>	failed	simple_python_operator	02-25T06:02:54.809486+00:00	manual_2020-02-25T06:02:54.809486+00:00	<input checked="" type="checkbox"/>
<input type="checkbox"/>	success	simple_python_operator	02-25T05:42:19.997484+00:00	manual_2020-02-25T05:42:19.997484+00:00	<input checked="" type="checkbox"/>
<input type="checkbox"/>	failed	simple_python_operator	02-25T05:39:23.812012+00:00	manual_2020-02-25T05:39:23.812012+00:00	<input checked="" type="checkbox"/>
<input type="checkbox"/>	success	simple_python_operator	02-25T05:29:00.355729+00:00	manual_2020-02-25T05:29:00.355729+00:00	<input checked="" type="checkbox"/>
<input type="checkbox"/>	success	simple_python_operator	02-25T05:25:40.655538+00:00	manual_2020-02-25T05:25:40.655538+00:00	<input checked="" type="checkbox"/>
<input type="checkbox"/>	success	simple_python_operator	02-25T05:21:10.765962+00:00	manual_2020-02-25T05:21:10.765962+00:00	<input checked="" type="checkbox"/>

DAG Runs state



The screenshot shows the Airflow web interface for managing DAG runs. At the top, there's a navigation bar with links for Airflow, DAGs, Data Profiling, Browse, Admin, Docs, and About, along with a timestamp of 2020-02-26 14:31:13 UTC.

The main section is titled "Dag Runs" and displays a table of 12 entries. The table has columns for selecting a run (checkbox), State, Dag Id, Execution Date, Run Id, and External Trigger. The "State" column is highlighted with a red border. The "Dag Id" column shows "simple_python_operator" for all runs. The "Execution Date" column shows various dates from 2020-02-25T06:39:50 to 2020-02-25T05:21:10. The "Run Id" column shows manual identifiers corresponding to each date. The "External Trigger" column contains checked checkboxes for all runs.

<input type="checkbox"/>	State	Dag Id	Execution Date	Run Id	External Trigger
<input type="checkbox"/>	failed	simple_python_operator	02-25T06:39:50.248084+00:00	manual_2020-02-25T06:39:50.248084+00:00	<input checked="" type="checkbox"/>
<input type="checkbox"/>	success	simple_python_operator	02-25T06:08:08.823092+00:00	manual_2020-02-25T06:08:08.823092+00:00	<input checked="" type="checkbox"/>
<input type="checkbox"/>	failed	simple_python_operator	02-25T06:02:54.809486+00:00	manual_2020-02-25T06:02:54.809486+00:00	<input checked="" type="checkbox"/>
<input type="checkbox"/>	success	simple_python_operator	02-25T05:42:19.997484+00:00	manual_2020-02-25T05:42:19.997484+00:00	<input checked="" type="checkbox"/>
<input type="checkbox"/>	failed	simple_python_operator	02-25T05:39:23.812012+00:00	manual_2020-02-25T05:39:23.812012+00:00	<input checked="" type="checkbox"/>
<input type="checkbox"/>	success	simple_python_operator	02-25T05:29:00.355729+00:00	manual_2020-02-25T05:29:00.355729+00:00	<input checked="" type="checkbox"/>
<input type="checkbox"/>	success	simple_python_operator	02-25T05:25:40.655538+00:00	manual_2020-02-25T05:25:40.655538+00:00	<input checked="" type="checkbox"/>
<input type="checkbox"/>	success	simple_python_operator	02-25T05:21:10.765962+00:00	manual_2020-02-25T05:21:10.765962+00:00	<input checked="" type="checkbox"/>

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:

- @hourly
- @daily
- @weekly
- @monthly
- @yearly

cron equivalent:

- `0 * * * *`
- `0 0 * * *`
- `0 0 * * 0`
- `0 0 1 * *`
- `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!

INTRODUCTION TO AIRFLOW IN PYTHON

Airflow sensors

INTRODUCTION TO AIRFLOW IN PYTHON



Mike Metzger
Data Engineer

Sensors

What is a *sensor*?

- An operator that waits for a certain condition to be true
 - Creation of a file
 - Upload of a database record
 - Certain response from a web request
- Can define how often to check for the condition to be true
- Are assigned to tasks

Sensor details

- Derived from `airflow.sensors.base_sensor_operator`
- Sensor arguments:
 - `mode` - How to check for the condition
 - `mode='poke'` - The default, run repeatedly
 - `mode='reschedule'` - Give up task slot and try again later
 - `poke_interval` - How often to wait between checks
 - `timeout` - How long to wait before failing task
- Also includes normal operator attributes

File sensor

- Is part of the `airflow.contrib.sensors` library
- Checks for the existence of a file at a certain location
- Can also check if any files exist within a directory

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

file_sensor_task = FileSensor(task_id='file_sense',
                               filepath='salesdata.csv',
                               poke_interval=300,
                               dag=sales_report_dag)

init_sales_cleanup >> file_sensor_task >> generate_report
```

Other sensors

- `ExternalTaskSensor` - wait for a task in another DAG to complete
- `HttpSensor` - Request a web URL and check for content
- `SqlSensor` - Runs a SQL query to check for content
- Many others in `airflow.sensors` and `airflow.contrib.sensors`

Why sensors?

Use a sensor...

- Uncertain when it will be true
- If failure not immediately desired
- To add task repetition without loops

Let's practice!

INTRODUCTION TO AIRFLOW IN PYTHON

Airflow executors

INTRODUCTION TO AIRFLOW IN PYTHON



Mike Metzger
Data Engineer

What is an executor?

- Executors run tasks
- Different executors handle running the tasks differently
- Example executors:
 - SequentialExecutor
 - LocalExecutor
 - CeleryExecutor

SequentialExecutor

- The default Airflow executor
- Runs one task at a time
- Useful for debugging
- While functional, not really recommended for production

LocalExecutor

- Runs on a single system
- Treats tasks as processes
- *Parallelism* defined by the user
- Can utilize all resources of a given host system

CeleryExecutor

- Uses a Celery backend as task manager
- Multiple worker systems can be defined
- Is significantly more difficult to setup & configure
- Extremely powerful method for organizations with extensive workflows

Determine your executor

- Via the `airflow.cfg` file
- Look for the `executor=` line

```
repl:~$ cat airflow/airflow.cfg | grep "executor = "
executor = SequentialExecutor
repl:~$ █
```

Determine your executor #2

- Via the first line of `airflow list_dags`
- INFO - Using SequentialExecutor

```
repl:~$ airflow list_dags
[2020-04-05 19:29:37,647] {__init__.py:51} INFO - Using executor SequentialExecutor
[2020-04-05 19:29:37,973] {dagbag.py:90} INFO - Filling up the DagBag from /home/repl/workspace/dags
```

Let's practice!

INTRODUCTION TO AIRFLOW IN PYTHON

Debugging and troubleshooting in Airflow

INTRODUCTION TO AIRFLOW IN PYTHON

Mike Metzger
Data Engineer



Typical issues...

- DAG won't run on schedule
- DAG won't load
- Syntax errors

DAG won't run on schedule

- Check if scheduler is running

The scheduler does not appear to be running. Last heartbeat was received 31 seconds ago.

The DAGs list may not update, and new tasks will not be scheduled.

DAGs

Search:

	i	DAG	Schedule	Owner	Recent Tasks i	Last Run i
		chained_operators	1 day, 0:00:00	chained_bag		
		example_bash_operator	0 0 * * *	Airflow		
		example_branch_dop_operator_v3	*/1 * * * *	Airflow		
		example_branch_operator	@daily	Airflow		
		example_http_operator	1 day, 0:00:00	Airflow		
		example_passing_params_via_test_command	*/1 * * * *	airflow		

- Fix by running `airflow scheduler` from the command-line.

DAG won't run on schedule

- At least one `schedule_interval` hasn't passed.
 - Modify the attributes to meet your requirements.
- Not enough tasks free within the executor to run.
 - Change executor type
 - Add system resources
 - Add more systems
 - Change DAG scheduling

DAG won't load

- DAG not in web UI
- DAG not in `airflow list_dags`

Possible solutions

- Verify DAG file is in correct folder
- Determine the DAGs folder via `airflow.cfg`
- Note, the folder must be an absolute path

```
repl:~$ head airflow/airflow.cfg
[core]
# The folder where your airflow pipelines live, most likely a
# subfolder in a code repository
# This path must be absolute
dags_folder = /home/repl/airflow/dags
```

Syntax errors

- The most common reason a DAG file won't appear
- Sometimes difficult to find errors in DAG
- Two quick methods:
 - Run `airflow list_dags`
 - Run `python3 <dagfile.py>`

airflow list_dags

```
repl:~/workspace$ airflow list_dags
[2020-04-08 04:05:55,369] {plugins_manager.py:148} ERROR - name 'BasOperator' is not defined
Traceback (most recent call last):
  File "/usr/local/lib/python3.6/dist-packages/airflow/plugins_manager.py", line 142, in <module>
    m = imp.load_source(namespace, filepath)
  File "/usr/lib/python3.6/imp.py", line 172, in load_source
    module = _load(spec)
  File "<frozen importlib._bootstrap>", line 684, in _load
  File "<frozen importlib._bootstrap>", line 665, in _load_unlocked
  File "<frozen importlib._bootstrap_external>", line 678, in exec_module
  File "<frozen importlib._bootstrap>", line 219, in _call_with_frames_removed
  File "/home/repl/workspace/dags/simple_dependency.py", line 13, in <module>
    task1 = BasOperator(task_id='first_task',
NameError: name 'BasOperator' is not defined
[2020-04-08 04:05:55,370] {plugins_manager.py:149} ERROR - Failed to import plugin /home/repl/workspace/dags/simple_dependency.py
```

Running the Python interpreter

```
python3 dagfile.py :
```

- With errors

```
(af) mmetzger@hugo:~/airflow/dags$ python3 simple_dependency.py
Traceback (most recent call last):
  File "simple_dependency.py", line 13, in <module>
    task1 = BasOperator(task_id='first_task',
NameError: name 'BasOperator' is not defined
```

- Without errors

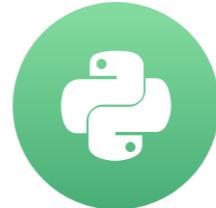
```
(af) mmetzger@hugo:~/airflow/dags$ python3 simple_python_operator.py
(af) mmetzger@hugo:~/airflow/dags$
```

Let's practice!

INTRODUCTION TO AIRFLOW IN PYTHON

SLAs and reporting in Airflow

INTRODUCTION TO AIRFLOW IN PYTHON



Mike Metzger
Data Engineer

SLAs

What is an SLA?

- An SLA stands for *Service Level Agreement*
- Within Airflow, the amount of time a task or a DAG should require to run
- An *SLA Miss* is any time the task / DAG does not meet the expected timing
- If an SLA is missed, an email is sent out and a log is stored.
- You can view SLA misses in the web UI.

SLA Misses

- Found under Browse: SLA Misses

Sla Misses

List (6) Add Filter ▾ Search: dag_id, task_id

	Dag Id	Task Id	Execution Date	Email Sent	Timestamp
	sla_test	failme ▾	04-08T06:07:00.568903+00:00	-	04-08T06:19:39.742894+00:00
	sla_test	failme ▾	04-08T06:09:00.568903+00:00	-	04-08T06:19:39.742894+00:00
	sla_test	failme ▾	04-08T06:11:00.568903+00:00	-	04-08T06:19:39.742894+00:00
	sla_test	failme ▾	04-08T06:13:00.568903+00:00	-	04-08T06:19:39.742894+00:00
	sla_test	failme ▾	04-08T06:15:00.568903+00:00	-	04-08T06:19:39.742894+00:00

Defining SLAs

- Using the `'sla'` argument on the task

```
task1 = BashOperator(task_id='sla_task',  
                     bash_command='runcode.sh',  
                     sla=timedelta(seconds=30),  
                     dag=dag)
```

- On the `default_args` dictionary

```
default_args={  
    'sla': timedelta(minutes=20)  
    'start_date': datetime(2020, 2, 20)  
}  
dag = DAG('sla_dag', default_args=default_args)
```

timedelta object

- In the `datetime` library
- Accessed via `from datetime import timedelta`
- Takes arguments of days, seconds, minutes, hours, and weeks

```
timedelta(seconds=30)
```

```
timedelta(weeks=2)
```

```
timedelta(days=4, hours=10, minutes=20, seconds=30)
```

General reporting

- Options for success / failure / error
- Keys in the default_args dictionary

```
default_args={  
    'email': ['airflowalerts@datacamp.com'],  
    'email_on_failure': True,  
    'email_on_retry': False,  
    'email_on_success': True,  
    ...  
}
```

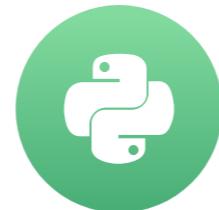
- Within DAGs from the EmailOperator

Let's practice!

INTRODUCTION TO AIRFLOW IN PYTHON

Working with templates

INTRODUCTION TO AIRFLOW IN PYTHON



Mike Metzger
Data Engineer

What are templates?

Templates:

- Allow substituting information during a DAG run
- Provide added flexibility when defining tasks
- Are created using the `Jinja` templating language

Non-Templated BashOperator example

Create a task to echo a list of files:

```
t1 = BashOperator(  
    task_id='first_task',  
    bash_command='echo "Reading file1.txt"',  
    dag=dag)  
  
t2 = BashOperator(  
    task_id='second_task',  
    bash_command='echo "Reading file2.txt"',  
    dag=dag)
```

Templated BashOperator example

```
templated_command="""
    echo "Reading {{ params.filename }}"
"""

t1 = BashOperator(task_id='template_task',
                  bash_command=templated_command,
                  params={'filename': 'file1.txt'}
                  dag=example_dag)
```

Output:

```
Reading file1.txt
```

Templated BashOperator example (continued)

```
templated_command="""
    echo "Reading {{ params.filename }}"
"""

t1 = BashOperator(task_id='template_task',
                  bash_command=templated_command,
                  params={'filename': 'file1.txt'}
                  dag=example_dag)

t2 = BashOperator(task_id='template_task',
                  bash_command=templated_command,
                  params={'filename': 'file2.txt'}
                  dag=example_dag)
```

Let's practice!

INTRODUCTION TO AIRFLOW IN PYTHON

More templates

INTRODUCTION TO AIRFLOW IN PYTHON



Mike Metzger
Data Engineer

Quick task reminder

- Take a list of filenames
- Print "Reading <filename>" to the log / output
- Templated version:

```
templated_command="""
echo "Reading {{ params.filename }}"
"""

t1 = BashOperator(task_id='template_task',
                  bash_command=templated_command,
                  params={'filename': 'file1.txt'}
                  dag=example_dag)
```

More advanced template

```
templated_command="""
{% for filename in params.filenames %}
    echo "Reading {{ filename }}"
{% endfor %}
"""

t1 = BashOperator(task_id='template_task',
                  bash_command=templated_command,
                  params={'filenames': ['file1.txt', 'file2.txt']}
                  dag=example_dag)
```

```
Reading file1.txt
Reading file2.txt
```

Variables

- Airflow built-in runtime variables
- Provides assorted information about DAG runs, tasks, and even the system configuration.
- Examples include:

```
Execution Date: {{ ds }} # YYYY-MM-DD  
Execution Date, no dashes: {{ ds_nodash }} # YYYYMMDD  
Previous Execution date: {{ prev_ds }} # YYYY-MM-DD  
Prev Execution date, no dashes: {{ prev_ds_nodash }} # YYYYMMDD  
DAG object: {{ dag }}  
Airflow config object: {{ conf }}
```

¹ <https://airflow.apache.org/docs/stable/macros-ref.html>

Macros

In addition to others, there is also a `{{ macros }}` variable.

This is a reference to the Airflow macros package which provides various useful objects / methods for Airflow templates.

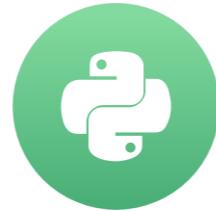
- `{{ macros.datetime }}` : The `datetime.datetime` object
- `{{ macros.timedelta }}` : The `timedelta` object
- `{{ macros.uuid }}` : Python's `uuid` object
- `{{ macros.ds_add('2020-04-15', 5) }}` : Modify days from a date, this example returns 2020-04-20

Let's practice!

INTRODUCTION TO AIRFLOW IN PYTHON

Branching

INTRODUCTION TO AIRFLOW IN PYTHON



Mike Metzger
Data Engineer

Branching

Branching in Airflow:

- Provides conditional logic
- Using `BranchPythonOperator`
- `from airflow.operators.python_operator import BranchPythonOperator`
- Takes a `python_callable` to return the next task id (or list of ids) to follow

Branching example

```
def branch_test(**kwargs):
    if int(kwargs['ds_nodash']) % 2 == 0:
        return 'even_day_task'
    else:
        return 'odd_day_task'
```

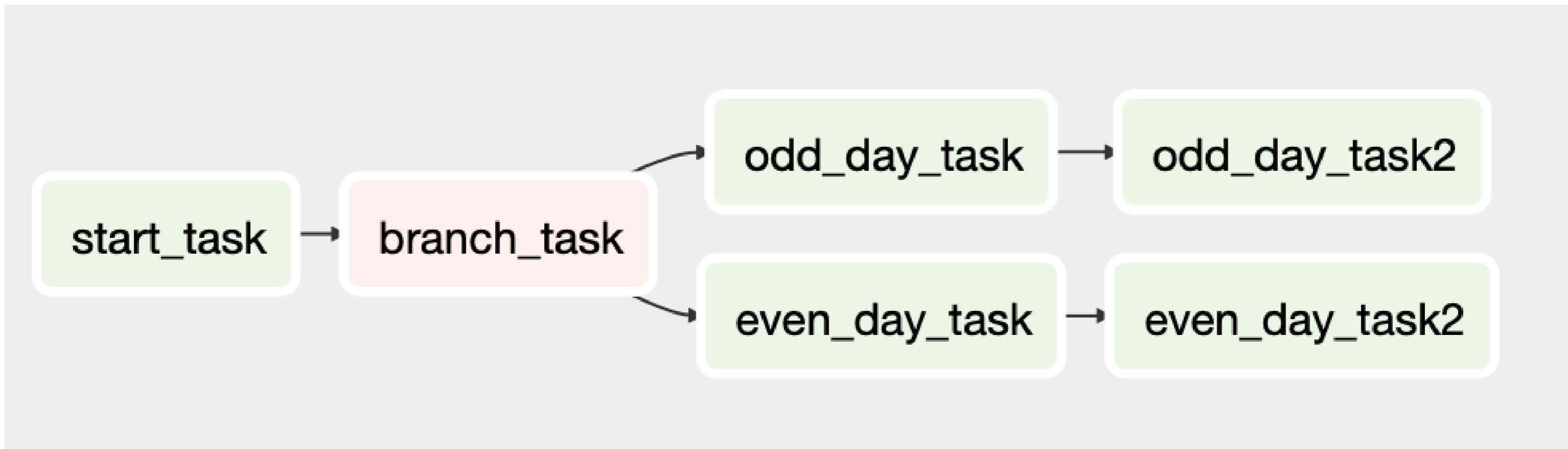
Branching example

```
def branch_test(**kwargs):
    if int(kwargs['ds_nodash']) % 2 == 0:
        return 'even_day_task'
    else:
        return 'odd_day_task'

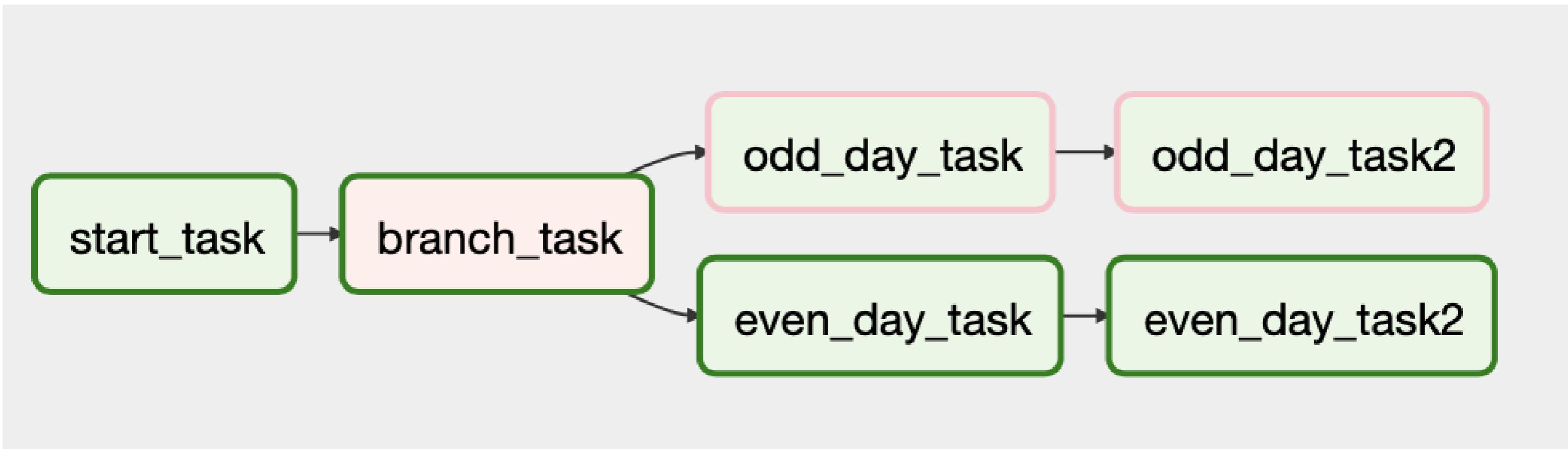
branch_task = BranchPythonOperator(task_id='branch_task', dag=dag,
                                   provide_context=True,
                                   python_callable=branch_test)

start_task >> branch_task >> even_day_task >> even_day_task2
branch_task >> odd_day_task >> odd_day_task2
```

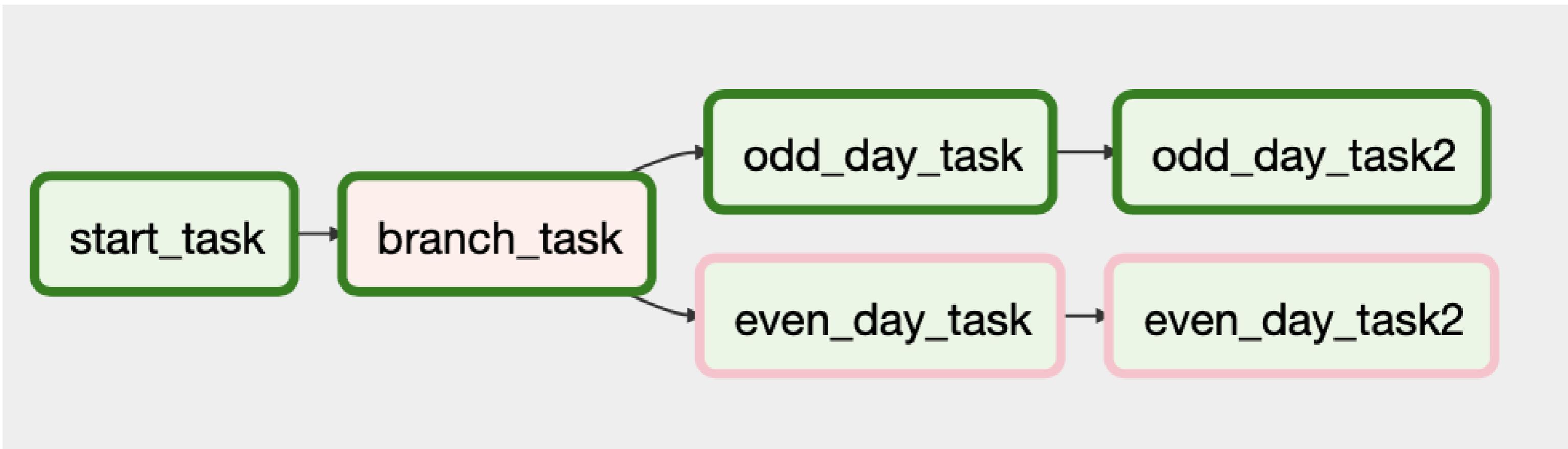
Branching graph view



Branching even days



Branching odd days

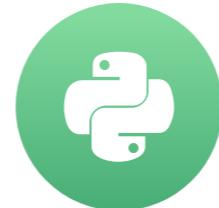


Let's practice!

INTRODUCTION TO AIRFLOW IN PYTHON

Creating a production pipeline

INTRODUCTION TO AIRFLOW IN PYTHON



Mike Metzger
Data Engineer

Running DAGs & Tasks

To run a specific task from command-line:

```
airflow run <dag_id> <task_id> <date>
```

To run a full DAG:

```
airflow trigger_dag -e <date> <dag_id>
```

Operators reminder

- BashOperator - expects a `bash_command`
- PythonOperator - expects a `python_callable`
- BranchPythonOperator - requires a `python_callable` and `provide_context=True`. The callable must accept `**kwargs`.
- FileSensor - requires `filepath` argument and might need `mode` or `poke_interval` attributes

Template reminders

- Many objects in Airflow can use templates
- Certain fields may use templated strings, while others do not
- One way to check is to use built-in documentation:
 1. Open python3 interpreter
 2. Import necessary libraries (ie,

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

)
 3. At prompt, run `help(<Airflow object>)`, ie, `help(BashOperator)`
 4. Look for a line that referencing `template_fields`. This will specify any of the arguments that can use templates.

Template documentation example

```
repl:~$ python3
Python 3.6.7 (default, Oct 22 2018, 11:32:17)
[GCC 8.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> from airflow.operators.bash_operator import BashOperator
>>> help(BashOperator)
```

Data and other attributes defined here:

```
template_ext = ('.sh', '.bash')
```

```
template_fields = ('bash_command', 'env')
```

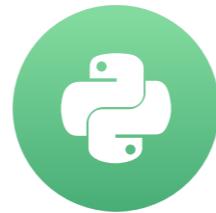
```
ui_color = '#f0e0e0'
```

Let's practice!

INTRODUCTION TO AIRFLOW IN PYTHON

Congratulations!

INTRODUCTION TO AIRFLOW IN PYTHON



Mike Metzger
Data Engineer

What we've learned

- Workflows / DAGs
- Operators (BashOperator, PythonOperator, EmailOperator)
- Tasks
- Dependencies / Bitshift operators
- Sensors
- Scheduling
- SLAs / Alerting
- Templates
- Branching
- Airflow command line / UI
- Airflow executors
- Debugging / Troubleshooting

Next steps

- Setup your own environment for practice
- Look into other operators / sensors
- Experiment with dependencies
- Look into parts of Airflow we didn't cover
 - XCom
 - Connections
 - Refer to docs for more
- Keep building workflows!

Thank you!

INTRODUCTION TO AIRFLOW IN PYTHON