

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



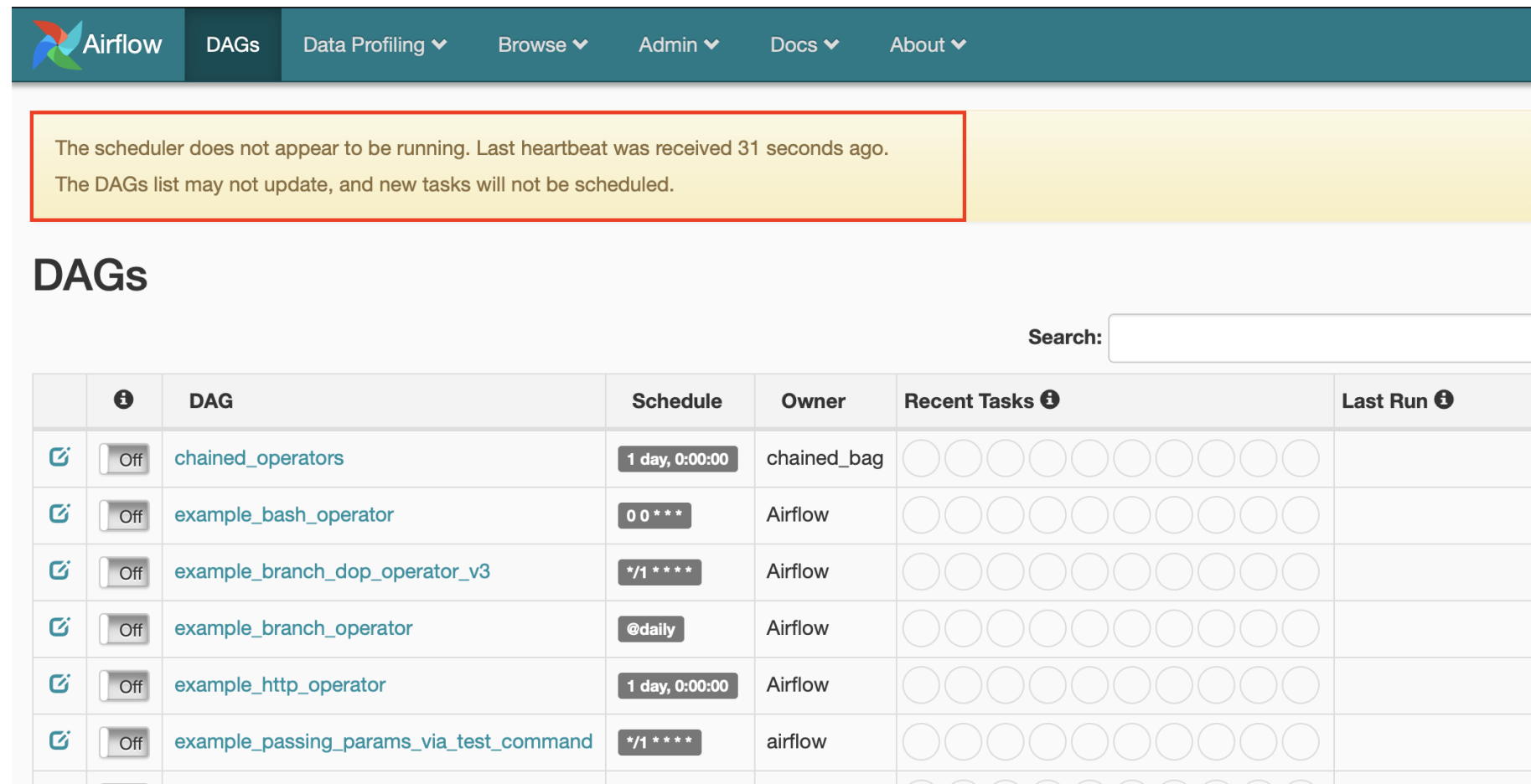
Mike Metzger
Data Engineer

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 screenshot shows the Airflow web interface. At the top, there's a navigation bar with the Airflow logo and links for DAGs, Data Profiling, Browse, Admin, Docs, and About. Below the navigation bar, a yellow warning box with a red border contains the message: "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." Below the warning box, the "DAGs" section is visible, featuring a search bar and a table of DAGs. The table has columns for DAG, Schedule, Owner, Recent Tasks, and Last Run. The DAGs listed are chained_operators, example_bash_operator, example_branch_dop_operator_v3, example_branch_operator, example_http_operator, and example_passing_params_via_test_command. Each DAG has a toggle switch set to "Off".

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

 Airflow

DAGs

Data Profiling ▾

Browse ▾

Admin ▾

Docs ▾

About ▾

2020-04-08 06:20:08 UTC

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