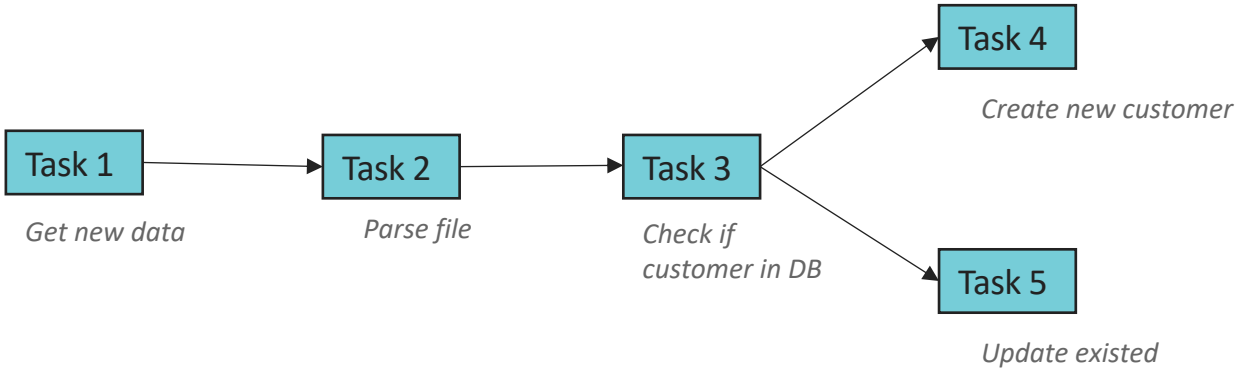
Apache Airflow, Day 2

Iuliia Volkova, xnuinside@gmail.com

Agenda (Day 2)

1. Macros, User Defined Marcos, Xcom
2. SLAs, Alerts, Retries
3. BranchOperator, TriggerRules
4. Hooks, Connections
5. Executors
6. Configuration (let's add Celery Executor & PostgreSQL)
7. Workers & Flower
8. Variables, Run DAG with Params
9. Backfill
10. Customization: UI plugins
11. Airflow in clouds: Google Compose (Airflow in GCP), Astronomer.io
12. Q&A session

DAG – Directed Acyclic Graph



What is a **Task**

- **Action** to do
- Process to **execute**
- Some atomic step of work that must be done

Example of task

- **Read** the file
- **Execute** the SQL query
- **Upload** the file
- **Download** the file
- etc

Type of tasks in Airflow

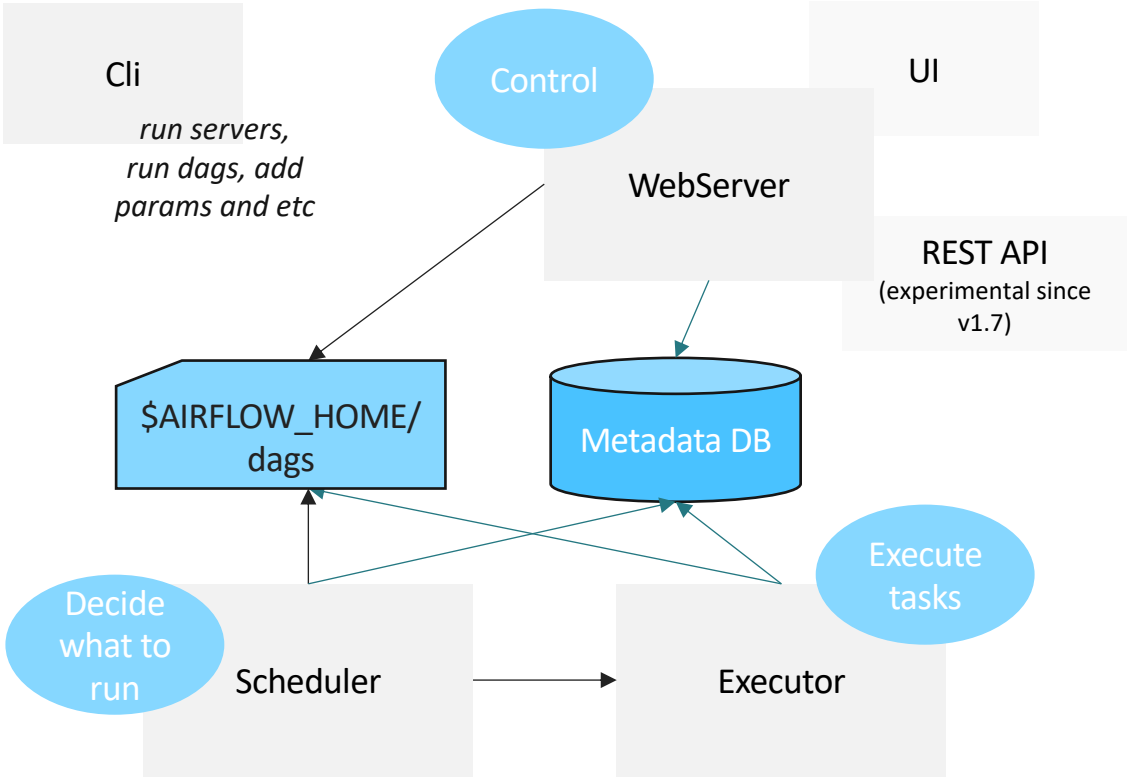
- Operator – DO something
- Sensor – wait until something will be True

FileSensor

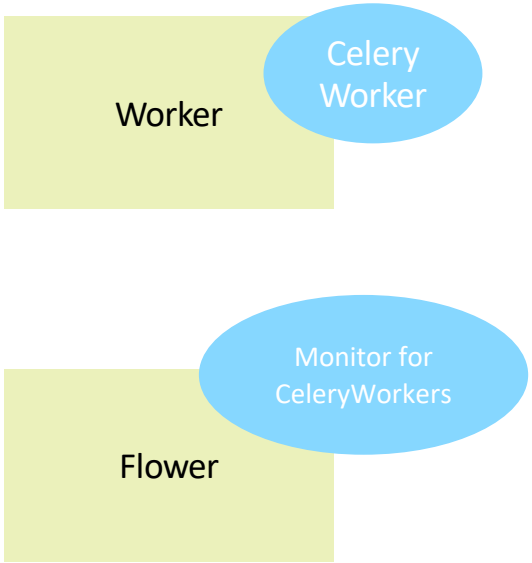
```
from datetime import datetime
from airflow import DAG
from airflow.operators.dummy_operator import DummyOperator
from airflow.contrib.sensors.file_sensor import FileSensor


with DAG(
  dag_id="consume_new_data_from_pos_read_and_parse",
  start_date=datetime(2020, 12, 1),
  schedule_interval="0 * * * *"
  ) as dag:
  get_new_data = FileSensor(task_id="get_new_data",
                            filepath="../shop123/${current_date}/${hour}/data.json")
```

If File ="../shop123/${current_date}/${hour}/data.json" was found – Run next task

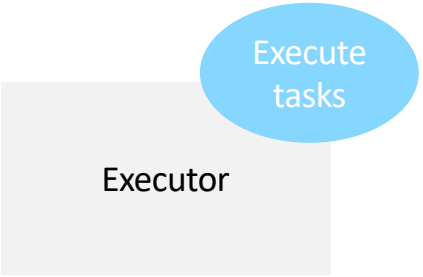
High-level overview of Apache Airflow components



If you work with CeleryExecutor



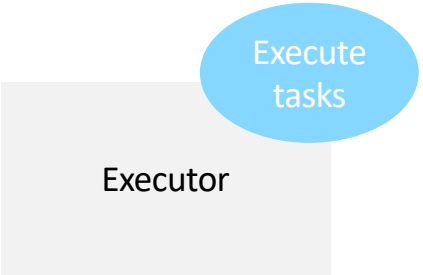
Executors



- Sequential Executor
- Debug Executor (can be used from IDE for debug)
- Local Executor (parallel execution with Python multiprocessing)

https://airflow.apache.org/docs/apache-airflow/stable/executor/index.html

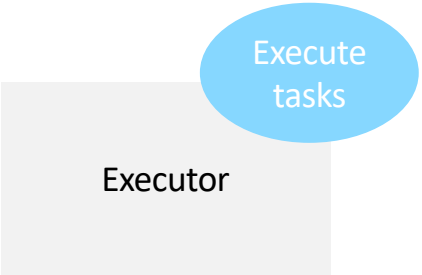
Executors



- Sequential Executor (parallel execution with Python multiprocessing)
- Debug Executor (can be used from IDE for debug)
- Local Executor (parallel execution with Python multiprocessing)
- Dask Executor - https://dask.org/

https://airflow.apache.org/docs/apache-airflow/stable/executor/index.html

Executors



- Sequential Executor (parallel execution with Python multiprocessing)
- Debug Executor (can be used from IDE for debug)
- Local Executor (parallel execution with Python multiprocessing)
- Dask Executor - https://dask.org/
- Celery Executor
- **Kubernetes** Executor
- Scaling Out with Mesos (community contributed)

https://airflow.apache.org/docs/apache-airflow/stable/executor/index.html

Celery Executor

https://docs.celeryproject.org/en/stable/getting-started/introduction.html

Python Open Source Server for distributing work across threads or machines.

https://github.com/xnuinside/airflow_in_docker_compose – we will use Docker Compose

Airflow.cfg

https://airflow.apache.org/docs/apache-airflow/stable/configurations-ref.html

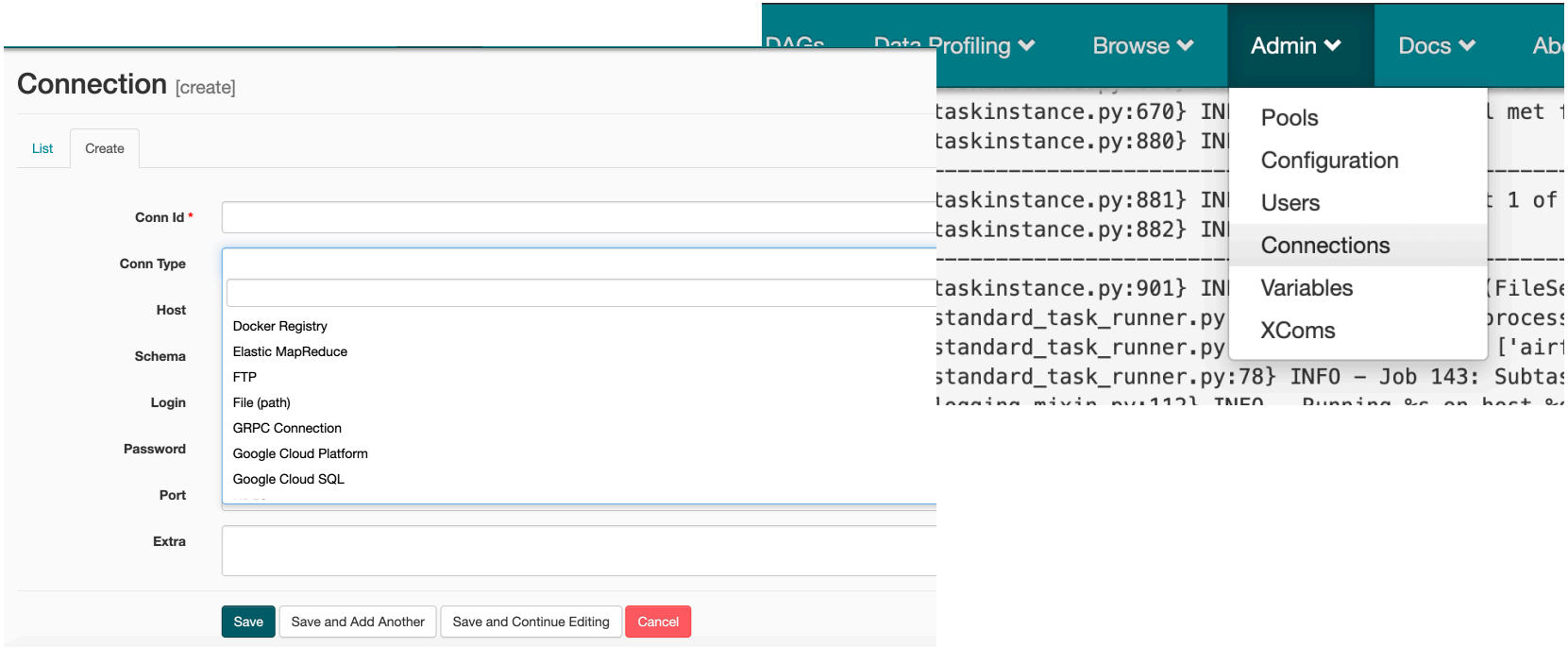
Need to setup executor= & sql_alchemy_conn=

Default Connections (no exists in Docker version)

```
  File "/home/airflow/.local/lib/python3.6/site-packages/airflow/contrib/hooks/fs_hook.py", line 38, in __init__
    conn = self.get_connection(conn_id)
  File "/home/airflow/.local/lib/python3.6/site-packages/airflow/hooks/base_hook.py", line 87, in get_connection
    conn = random.choice(list(cls.get_connections(conn_id)))
  File "/home/airflow/.local/lib/python3.6/site-packages/airflow/hooks/base_hook.py", line 83, in get_connections
    return secrets.get_connections(conn_id)
  File "/home/airflow/.local/lib/python3.6/site-packages/airflow/secrets/__init__.py", line 59, in get_connections
    raise AirflowException("The conn_id `{0}` isn't defined".format(conn_id))
airflow.exceptions.AirflowException: The conn_id `fs_default` isn't defined
[2020-12-09 14:01:21,102] {taskinstance.py:1194} INFO - Marking task as FAILED. dag_id=custom_macros_file_sensor_consume
[2020-12-09 14:01:21,425] {local_task_job.py:159} WARNING - State of this instance has been externally set to failed. Ta
[2020-12-09 14:01:21,436] {helpers.py:325} INFO - Sending Signals.SIGTERM to GPID 388
```

Connections & Hooks

https://airflow.apache.org/docs/apache-airflow/1.10.12/howto/connection/index.html



Connections & Hooks

Hooks defines API how to connect/work/talk with third-party systems

One **Hook** can be used in Multiple **Operators/Sensors**

Connections & Hooks

Hooks defines API how to connect/work/talk with third-party systems

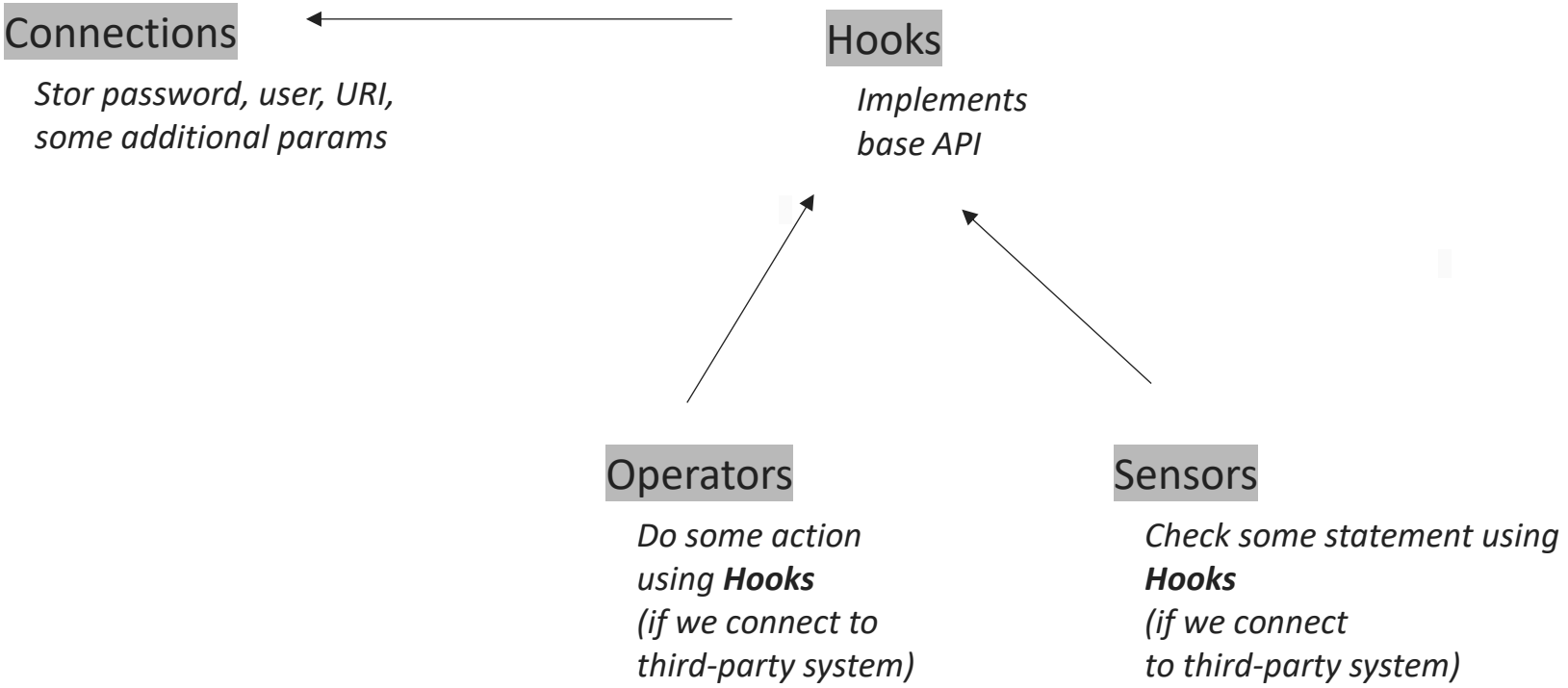
One **Hook** can be used in Multiple **Operators/Sensors**

For example:

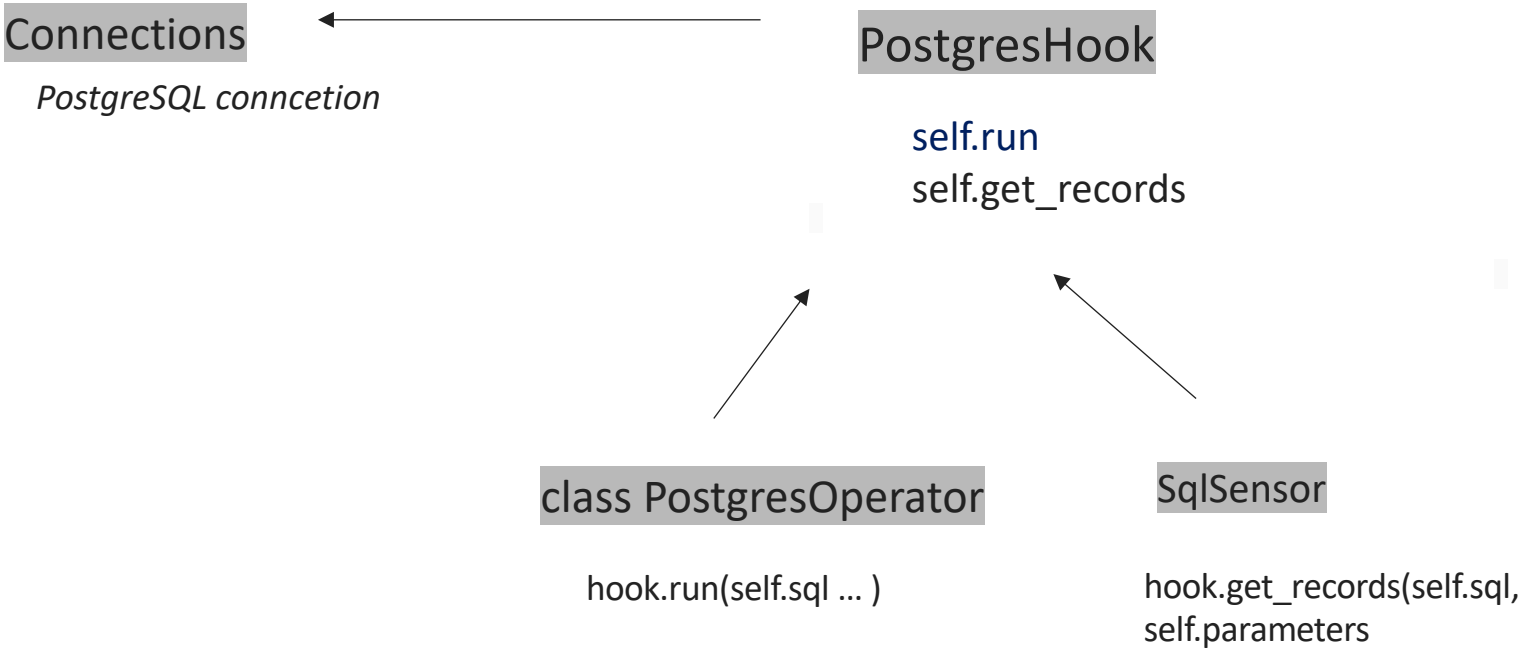
MySqlHook(DbApiHook) *implements:*
get_conn, get_autocommit, get_iam_token

DbApiHook implements:
get_sqlalchemy_engine
get_records
get_first & etc.

Connections -> Hooks -> Operators/Sensors



Example


Connections
PostgreSQL conncetion
PostgresHook
self.run
self.get_records
class PostgresOperator
hook.run(self.sql ...)
SqlSensor
hook.get_records(self.sql,
self.parameters

Let’s add fs_default connection

Connections

List (1) Create With selected

		Conn Id	Conn Type	Host	Port	Is Encrypted	Is Extra Encrypted
☐		fs_default	fs			⊖	⊖

Macros vs Jinja2 Template

```
"../shop123/${current_date}/${hour}/data.json”
```

Parameters
that depends on DAG Run (when we execute our pipeline):

${current_date} – {{ ds_nodash }} (variable from - https://airflow.apache.org/docs/apache-airflow/1.10.8/macros.html)

${hour} - {{ ts_nodash.split('T')[1][:2] }} **- ts_nodash returns 20180101T000000**

Macros vs Jinja2 Template

```
"../shop123/${current_date}/${hour}/data.json”
```

Parameters
that depends on DAG Run (when we execute our pipeline):

${current_date} – {{ ds_nodash }} (variable from - https://airflow.apache.org/docs/apache-airflow/1.10.8/macros.html)

${hour} - {{ ts_nodash.split('T')[1][:2] }} **- ts_nodash returns 20180101T000000**

Parametrized path:

```
get_new_data = FileSensor(task_id="get_new_data",
                          filepath="../shop123/ {{ ds_nodash }} / {{ ts_nodash.split('T')[1][:2] }} /data.json")
```

Macros vs Jinja2 Template

```
020-12-09 15:44:08,309] {taskinstance.py:882} INFO -
--------------------------------------------------------------------------------
020-12-09 15:44:08,314] {taskinstance.py:901} INFO - Executing <Task(FileSensor): get_new_data> on 2020-12-01T02:00:00+00:00
020-12-09 15:44:08,317] {standard_task_runner.py:54} INFO - Started process 35930 to run task
020-12-09 15:44:08,364] {standard_task_runner.py:77} INFO - Running: ['airflow', 'run', 'file_sensor_consume_new_data', 'get_new_data', '2020-12-01T02:00:00+0
020-12-09 15:44:08,368] {standard_task_runner.py:78} INFO - Job 8: Subtask get_new_data
020-12-09 15:44:08,416] {logging_mixin.py:112} INFO - Running %s on host %s <TaskInstance: file_sensor_consume_new_data.get_new_data 2020-12-01T02:00:00+00:00
020-12-09 15:44:08,449] {file_sensor.py:60} INFO - Poking for file /../shop123/20201201/02/data.json
```

Macros vs Jinja2 Template

What if standard variables not enough?

Let’s imagine that **each shop has own pipeline (DAG).**

```
"../shop123/ {{ ds_nodash }} / {{ ts_nodash.split('T')[1][:2] }} /data.json”   ->
```

```
"../${shop}/ {{ ds_nodash }} / {{ ts_nodash.split('T')[1][:2] }} /data.json”
```

Macros vs Jinja2 Template

What if standard variables not enough?

Let’s imagine that **each shop has own pipeline (DAG).**

```
"../shop123/ {{ ds_nodash }} / {{ ts_nodash.split('T')[1][:2] }} /data.json”   ->

"../${shop}/ {{ ds_nodash }} / {{ ts_nodash.split('T')[1][:2] }} /data.json”
```

Macros vs Jinja2 Template

What if standard variables not enough?

Let’s create the custom macros.

Macros vs Jinja2 Template

```
def shop_filepath_macros(shop_id, date, hour):
  file_path = f"./{shop_id}/{date}/{hour}/data.json"
  return file_path
```

Macros callable

```
with DAG(
  dag_id="custom_macros_file_sensor_consume_new_data",
  start_date=datetime(2020, 12, 1),
  schedule_interval="0 * * * *",
  user_defined_macros={
      'shop_filepath_macros': shop_filepath_macros
  }
  ) as dag:
  # task 1
  get_new_data = FileSensor(task_id="get_new_data",
                            filepath="{{ shop_filepath_macros('shop123', ds_nodash,
ts_nodash.split('T')[1][:2])}}")
```

Macros vs Jinja2 Template

```
def shop_filepath_macros(shop_id, date, hour):
  file_path = f"./{shop_id}/{date}/{hour}/data.json"
  return file_path


with DAG(
  dag_id="custom_macros_file_sensor_consume_new_data",
  start_date=datetime(2020, 12, 1),
  schedule_interval="0 * * * *",
  user_defined_macros={
      'shop_filepath_macros': shop_filepath_macros
  }
  ) as dag:
  # task 1
  get_new_data = FileSensor(task_id="get_new_data",
                            filepath="{{ shop_filepath_macros('shop123', ds_nodash,
ts_nodash.split('T')[1][:2])}}")
```

Define macros

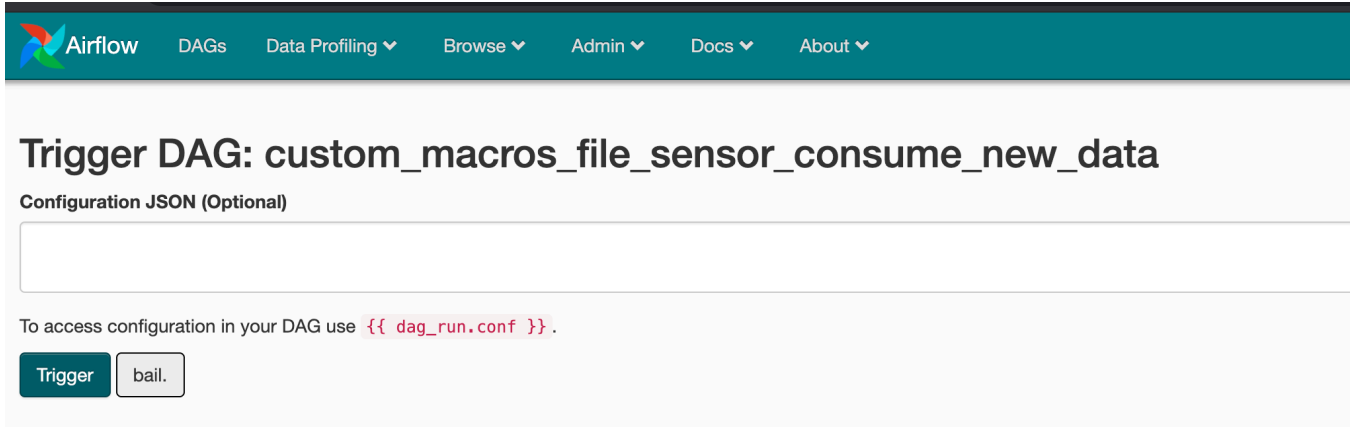
Macros vs Jinja2 Template

```
def shop_filepath_macros(shop_id, date, hour):
  current_dir = os.path.dirname(os.path.abspath(__file__))
  file_path = f"./{shop_id}/{date}/{hour}/data.json"
  return file_path


with DAG(
  dag_id="custom_macros_file_sensor_consume_new_data",
  start_date=datetime(2020, 12, 1),
  schedule_interval="0 * * * *",
  user_defined_macros={
      'shop_filepath_macros': shop_filepath_macros
  }
  ) as dag:
  # task 1
  get_new_data = FileSensor(task_id="get_new_data",
                            filepath="{{ shop_filepath_macros('shop123', ds_nodash,
ts_nodash.split('T')[1][:2])}}")
```

use macros

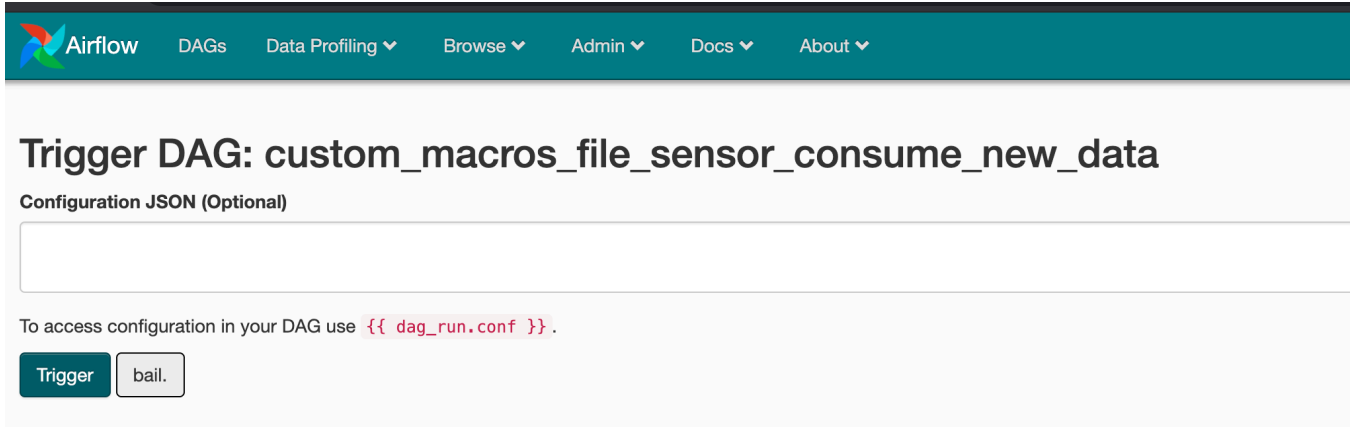
DAGRun Config

```
get_new_data = FileSensor(task_id="get_new_data",
                          filepath="{{ shop_filepath_macros('shop123',
ds_nodash, ts_nodash.split('T')[1][:2])}}")
```



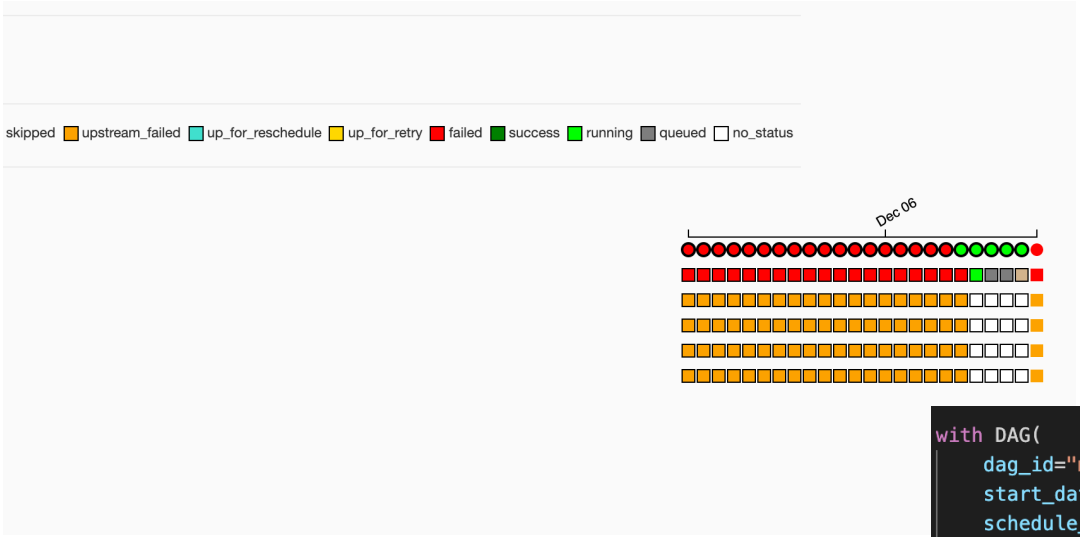
DAGRun Config

```
get_new_data = FileSensor(task_id="get_new_data",
                          filepath="{{ shop_filepath_macros('shop123',
ds_nodash, ts_nodash.split('T')[1][:2])}}")
```




```
airflow trigger_dag 'example_dag_conf' -r 'run_id' --conf '{"message":"value"}'
```

Max Active DAG Runs




```python
with DAG(
    dag_id="max_active_dag_run_1",
    start_date=datetime(2020, 12, 1),
    schedule_interval="0 * * * *",
    user_defined_macros={
        'shop_filepath_macros': shop_filepath_macros
    },
    max_active_runs=1
    ) as dag:
    # task 1
    get_new_data = FileSensor(task_id="get_new_data",
                              filepath="{{ shop_filepath
```

Check result with config trigger

```
--------------------------------------------------------------------------------
20-12-09 15:17:49,862] {taskinstance.py:901} INFO - Executing <Task(FileSensor): get_new_data> on 2020-12-09T15:16:13.921812+00:00
20-12-09 15:17:49,878] {standard_task_runner.py:54} INFO - Started process 2339 to run task
20-12-09 15:17:50,132] {standard_task_runner.py:77} INFO - Running: ['airflow', 'run', 'max_active_dag_run_1', 'get_new_data', '2020-12-09T1
20-12-09 15:17:50,140] {standard_task_runner.py:78} INFO - Job 388: Subtask get_new_data
20-12-09 15:17:50,404] {logging_mixin.py:112} INFO - Running %s on host %s <TaskInstance: max_active_dag_run_1.get_new_data 2020-12-09T15:16
20-12-09 15:17:50,585] {file_sensor.py:60} INFO - Poking for file /opt/airflow/dags/shop256/20201209/15/data.json
```

Xcom

Data exchange between tasks.

Points:

1. Xcom – table in DB
2. Less count of data – it’s not processing tool
3. Xcom by default pulled from last task

Python Operator

BranchOperator

Trigger Rules

https://airflow.apache.org/docs/apache-airflow/stable/concepts.html#trigger-rules

- **all_success:** (default) all parents have succeeded
- **all_failed:** all parents are in a failed or upstream_failed state
- **all_done:** all parents are done with their execution

Trigger Rules

https://airflow.apache.org/docs/apache-airflow/stable/concepts.html#trigger-rules

- **all_success:** (default) all parents have succeeded
- **all_failed:** all parents are in a failed or upstream_failed state
- **all_done:** all parents are done with their execution
- **one_failed:** fires as soon as at least one parent has failed, it does not wait for all parents to be done
- **one_success:** fires as soon as at least one parent succeeds, it does not wait for all parents to be done
- **none_failed:** all parents have not failed (failed or upstream_failed) i.e. all parents have succeeded or been skipped
- Etc.

Trigger Rules

https://airflow.apache.org/docs/apache-airflow/stable/concepts.html#trigger-rules

- **all_success:** (default) all parents have succeeded
- **all_failed:** all parents are in a failed or upstream_failed state
- **all_done:** all parents are done with their execution
- **one_failed:** fires as soon as at least one parent has failed, it does not wait for all parents to be done
- **one_success:** fires as soon as at least one parent succeeds, it does not wait for all parents to be done
- **none_failed:** all parents have not failed (failed or upstream_failed) i.e. all parents have succeeded or been skipped
- Etc.

Callbacks

Tasks Default Params

PostgreSQL Operator

Flower

Questions?