airflow-declarative Documentation

Release 1.1

Usermodel Team @ Rambler Digital Solutions

Sep 05, 2021

Contents:

Documentation https://airflow-declarative.readthedocs.io/

Source Code https://github.com/rambler-digital-solutions/airflow-declarative

Issue Tracker https://github.com/rambler-digital-solutions/airflow-declarative/issues

PyPI https://pypi.org/project/airflow-declarative/

Airflow Declarative allows to define Airflow DAGs declaratively with YAML.

An example of a simple declarative DAG:

```yaml
dags:
  my_dag:
    args:
      start_date: 2019-07-01
    operators:
      my_operator:
        class: airflow.operators.dummy_operator:DummyOperator
```

CHAPTER 1

Introduction

The primary way of constructing DAGs in Airflow is by describing them in Python: https://airflow.apache.org/tutorial.html.

airflow_declarative transforms the DAGs defined with YAML to their Python equivalents.

The public API is pretty simple: there are just 2 functions, both returning the `airflow.models.DAG` instances:

CHAPTER 2

Installation

2.1 Option 1: Upstream Airflow

This is a simplest option which you would probably want to go with if you're just evaluating Declarative.

The idea is simple:

1. Install *airflow_declarative* Python package (e.g. with `pip install airflow_declarative`)
2. In the Airflow's `dags_folder` create a new Python module (e.g. `declarative_dags.py`), which would expose the Declarative DAGs:

```
import os

import airflow_declarative

# Assuming that the yaml dags are located in the same directory
# as this Python module:
root = os.path.dirname(__file__)

dags_list = [
    airflow_declarative.from_path(os.path.join(root, item))
    for item in os.listdir(root)
    if item.endswith((".yml", ".yaml"))
]

globals().update({dag.dag_id: dag for dags in dags_list for dag in dags})
```

That's enough to get started.

None of the other Python DAGs would be affected in any way, so you can start gradually migrating to Declarative DAGs.

However, this approach has some shortcomings:

1. If any of the Declarative yamls would fail to load, none of the Declarative DAGs would get loaded (because the Python module would raise an exception during the DAG import phase)

2. There's a `dagbag_import_timeout` configuration setting in Airflow, which sets a timeout in seconds for a single Python DAG module import. This timeout would apply to loading all Declarative DAGs at once, which might be an issue if there are long commands in the dynamic *do*/*with_items* blocks.
3. The `Code` tab of a DAG in the web interface would show the source of the Python shim module instead of the DAGs' yamls.

2.2 Option 2: Patched Airflow

Airflow might be patched to have a built-in support for Declarative DAGs, which doesn't have the limitations listed above in the first option.

In this case, the Python shim is not needed. Airflow would natively load the yaml DAGs as if they were in Python.

We provide ready to use patches in the patches directory, which are named after an Airflow release that they were made for. Only one patch should be applied (the one that matches your Airflow version).

To apply a patch, you would need to build your own distribution of Airflow instead of installing one from PyPI.

To achieve that, the following (roughly) should be done:

```
AIRFLOW_VERSION=1.10.4
AIRFLOW_DEPS=celery,postgres

curl -o declarative.patch https://raw.githubusercontent.com/rambler-digital-solutions/
↪airflow-declarative/master/patches/${AIRFLOW_VERSION}.patch
pip download --no-binary=:all: --no-deps apache-airflow==${AIRFLOW_VERSION}
tar xzf apache-airflow-*.tar.gz
cd apache-airflow-*/
patch -p1 < ../declarative.patch
pip install ".[${AIRFLOW_DEPS}]"
```

CHAPTER 3

Declarative DAG Reference

The actual schema definition can be found in the airflow_declarative/schema.py module. Some examples of complete DAGs are also available in the tests/dags/good directory.

This document contains a verbose description of the declarative DAG schema.

The anatomy of a declarative DAG:

```yaml
# The comments below specify the names of the schema definition
# atoms which can be found in the `airflow_declarative/schema.py`
# module.

dags:
  my_dag:  # the DAG name

    defaults:
      sensors:
        # `SENSOR`
        args:
          # `SENSOR_ARGS`
          queue: my_sensors_queue
      operators:
        # `OPERATOR`
        args:
          # `OPERATOR_ARGS`
          queue: my_operators_queue

    args:
      # `DAG_ARGS`
      start_date: 2019-07-01
      schedule_interval: 1d
      default_args:
        # `SENSOR_ARGS` | `OPERATOR_ARGS`
        owner: my_name

    sensors:
```

(continues on next page)

(continued from previous page)

```yaml
      my_sensor:
        # `SENSOR`
        callback: myproject.mymodule:my_sensor_callback
        callback_args:
          my_kwarg: my_value
        args:
          # `SENSOR_ARGS`
          poke_interval: 1m

    operators:
      my_operator:
        # `OPERATOR`
        callback: myproject.mymodule:my_operator_callback
        callback_args:
          my_kwarg: my_value
        args:
          # `OPERATOR_ARGS`
          retries: 3

    flow:
      # `FLOW`
      my_sensor:
      - my_operator

    do:
    # `DO_TEMPLATE`
    - operators:
        my_crops_{{ item.name }}:
          # `OPERATOR`
          callback: myproject.myfruits:my_crops
          callback_args:
            fruit: '{{ item.fruit_props }}'
      # `sensors` can be used there too!
      flow:
        # `FLOW`
        my_operator:
        - my_crops_{{ item.name }}
      with_items:
      # `WITH_ITEMS`
      - name: pineapple
        fruit_props:
          shape: ellipsoid
          color: brown
      - name: watermelon
        fruit_props:
          shape: round
          color: green
```

3.1 DAG_ARGS

`DAG_ARGS` atom defines the `__init__` arguments of an Airflow DAG. The actual meaning of these args can be found in the `airflow.models.DAG` doc page.

3.2 OPERATOR / SENSOR

`OPERATOR` and `SENSOR` atoms look similarly, except that their `args` schemas are different. They both define an Airflow operator (note that Sensors in Airflow are considered to be Operators).

For an Operator, the `args` (the `OPERATOR_ARGS` atom) are the `__init__` args of the `airflow.models.BaseOperator`.

For a Sensor, the `args` (the `SENSOR_ARGS` atom) are the `__init__` args of the `airflow.sensors.base_sensor_operator.BaseSensorOperator`.

The `OPERATOR`/`SENSOR` callable might be specified as a class. Example for `airflow.operators.bash_operator.BashOperator`:

```
class: airflow.operators.bash_operator:BashOperator
args:
  bash_command: 'echo "Hello World {{ ds }}"'
```

... or as a Python callable:

```
callback: myproject.mymodule:my_operator_callback
callback_args:
  my_kwarg: my_value
args:
  retries: 3
```

If `callback` value is a function, then it should look like this:

```
def my_operator_callback(context, my_kwarg):
    print("Execution date", conext["ds"])
    print("my_kwarg", my_kwarg)
```

The `callback` might also be a class:

```
class MyOperatorCallback:
    def __init__(self, context, my_kwarg):
        self.ds = context["ds"]
        self.my_kwarg = my_kwarg

    def __call__(self):
        print("Execution date", self.ds)
        print("my_kwarg", self.my_kwarg)
```

`callback_args` key is relevant only when `callback` is used (i.e. it cannot be defined with `class`). The distinction between the `args` and the `callback_args` is simple:

- `args` are the `__init__` args for the `airflow.models.BaseOperator`, which is used under the hood to wrap the `callback`;
- `callback_args` are the additional kwargs which would be passed to the `callback` along with the task `context`.

3.3 default_args / defaults

`default_args` is a standard `airflow.models.DAG` `__init__` arg which specifies the default args of a `airflow.models.BaseOperator`. These args would be supplied to all DAG's operators and sensors.

The `defaults` dict is a Declarative's extension which allows to specify the args more granularly: only to `sensors` or only to `operators` (note that defaults specified in `operators` would not be applied to sensors).

3.4 FLOW

The `FLOW` atom defines the DAG links between the operators.

`FLOW` is a dict of lists, where a key is a downstream operator name, and a value is a list of upstream operators.

Consider the following flow:

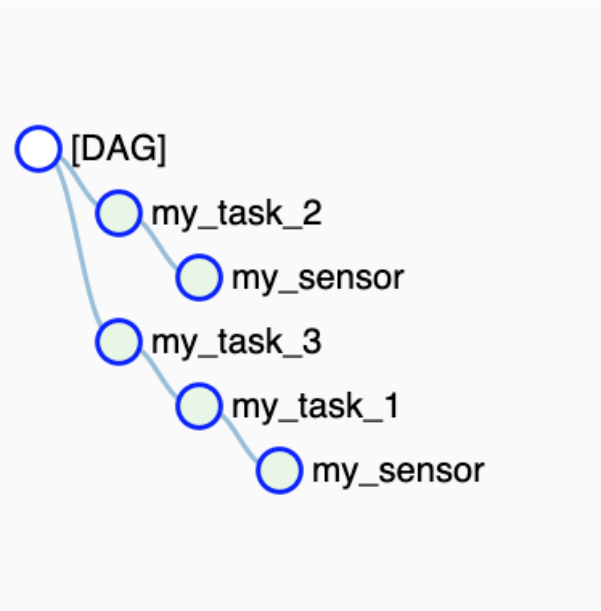
```
my_sensor:
- my_task_1
- my_task_2

my_task_1:
- my_task_3
```

Assuming that the Airflow operators are assigned to variables, the Python equivalent would be:

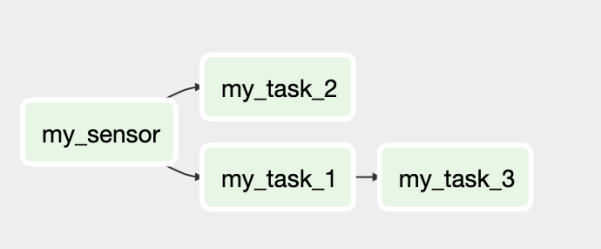
```
my_sensor.set_upstream(my_task_1)
my_sensor.set_upstream(my_task_2)

my_task_1.set_upstream(my_task_3)
```

This would be rendered in the Airflow web-interface like this:

- Tree view:



- Graph view:



3.5 DO (with_items)

The `do` block allows to make the DAG schema dynamic.

A `do` value is a list of dicts, each dict (a `DO_TEMPLATE`) must contain a `with_items` key and might optionally contain `operators`, `sensors` and `flow` – these have the same schema as the corresponding keys of the DAG.

`with_items` defines a list of items, which should be used to render a single `DO_TEMPLATE` block. Operators, Sensors and Flow within the block would be merged together (as dict unions).

There're 3 different ways to define `with_items`:

1. As a static list of items:

```
with_items:
- some_name: John
- some_name: Jill
```

2. As a Python callback, which returns a list of items:

```
with_items:
  using: myproject.mymodule:my_with_items
```

   Where `my_with_items` is a Python function which might look like this:

```
def my_with_items():
    return [
        {"some_name": "John"},
        {"some_name": "Jill"},
    ]
```

3. As an external program, which prints to stdout a list of items in JSON:

```
with_items:
  from_stdout: my_command --my-arg 42
```

   Where `my_command` is an executable in `$PATH`, which might look like this:

```
#!/usr/bin/env ruby

require 'json'

print [
  {some_name: "John"},
  {some_name: "Jill"},
].to_json
```

`operators`, `sensors` and `flow` within the `DO_TEMPLATE` block should use Jinja2 templates to render the items.

The following DAG defined by a `do` block:

```
operators:
  my_operator:
    callback: myproject.mymodule:my_operator_callback
do:
- operators:
    my_crops_{{ item.name }}:
      callback: myproject.myfruits:my_crops
      callback_args:
        fruit: '{{ item.fruit_props }}'
  flow:
    my_operator:
    - my_crops_{{ item.name }}
  with_items:
  - name: pineapple
    fruit_props:
      shape: ellipsoid
      color: brown
  - name: watermelon
    fruit_props:
      shape: round
      color: green
```

... is equivalent to the following DAG defined statically:

```
operators:
  my_operator:
    callback: myproject.mymodule:my_operator_callback
  my_crops_pineapple:
    callback: myproject.myfruits:my_crops
    callback_args:
      fruit:
        shape: ellipsoid
        color: brown
  my_crops_watermelon:
    callback: myproject.myfruits:my_crops
    callback_args:
      fruit:
        shape: round
        color: green
flow:
 my_operator:
 - my_crops_pineapple
 - my_crops_watermelon
```

CHAPTER 4

Testing Declarative DAGs

Unlike DAGs defined with Python, declarative DAGs forcibly draw a strong line between the business logic (the Python callbacks and external commands) and the DAG definition. This obligatory separation brings some benefits:

- The business logic doesn't have to know anything about Airflow existence (except the *task context dict*).
- The declaratively defined DAG doesn't have to be executed by Airflow. A custom implementation could be created which could execute the DAGs described with YAML, which is easy to parse.
- It is much clearer what has to be tested and what is not.

The business logic is completely under your responsibility so it should be covered with tests.

The DAG instantiation, however, is not your responsibility, so it doesn't have to be covered with tests. It is just enough to ensure that the declarative DAG passes the schema validation and looks sensible (i.e. all required operators and sensors are defined, the *flow* contains all required links).

The schema validation could be automated with the following test:

```
from pathlib import Path

import airflow_declarative
import pytest
from airflow import DAG


DAG_DIR = Path("share") / "airflow"


@pytest.mark.parametrize("dag_path", DAG_DIR.glob("*.yaml"))
def test_load_airflow_dags(dag_path):
    dags = airflow_declarative.from_path(dag_path)
    assert all(isinstance(dag, DAG) for dag in dags)
```

This test assumes that all of your declarative DAGs are located in the `share/airflow` directory. It loads all yamls from that directory, validates their schemas and transforms them to `airflow.models.DAG` instances. If a declarative DAG passes this test, then it can be loaded by the actual Airflow.

CHAPTER 5

Indices and tables

- genindex
- search