Airflow: Lesser Known Tips, Tricks, and Best Practises

Kaxil Naik
Dec 25, 2018 · 5 min read

There are certain things with all the tools you use that you won't know even after using it for a long time. And once you know it you are like "I wish I knew this before" as you had already told your client that it can't be done in any better way 🤦🤦. **Airflow** like other tool is no different, there are some hidden gems that can make your life easy and make **DAG** development fun.

You might already know some of them and if you know them all — well you are a PRO then💂🎩.

(1) DAG with context Manager

Were you annoyed with yourself when you forgot to add `dag=dag` to your task and Airflow error'ed? Yes, it is easy to forget adding it for each task. It is also redundant to

add the same parameter as shown in the following example (`example_dag.py` file):

```python
# Normal DAG without Context Manager
args = {
    'owner': 'airflow',
    'start_date': airflow.utils.dates.days_ago(2),
}

dag = DAG(
    dag_id='example_dag',
    default_args=args,
    schedule_interval='0 0 * * *',
)

run_this_last = DummyOperator(
    task_id='run_this_last',
    dag=dag,  # You need to repeat this for each task
)

run_this_first = BashOperator(
    task_id='run_this_first',
    bash_command='echo 1',
    dag=dag,  # You need to repeat this for each task
)

run_this_first >> run_this_last
```

example_dag.py hosted with ♡ by GitHub view raw

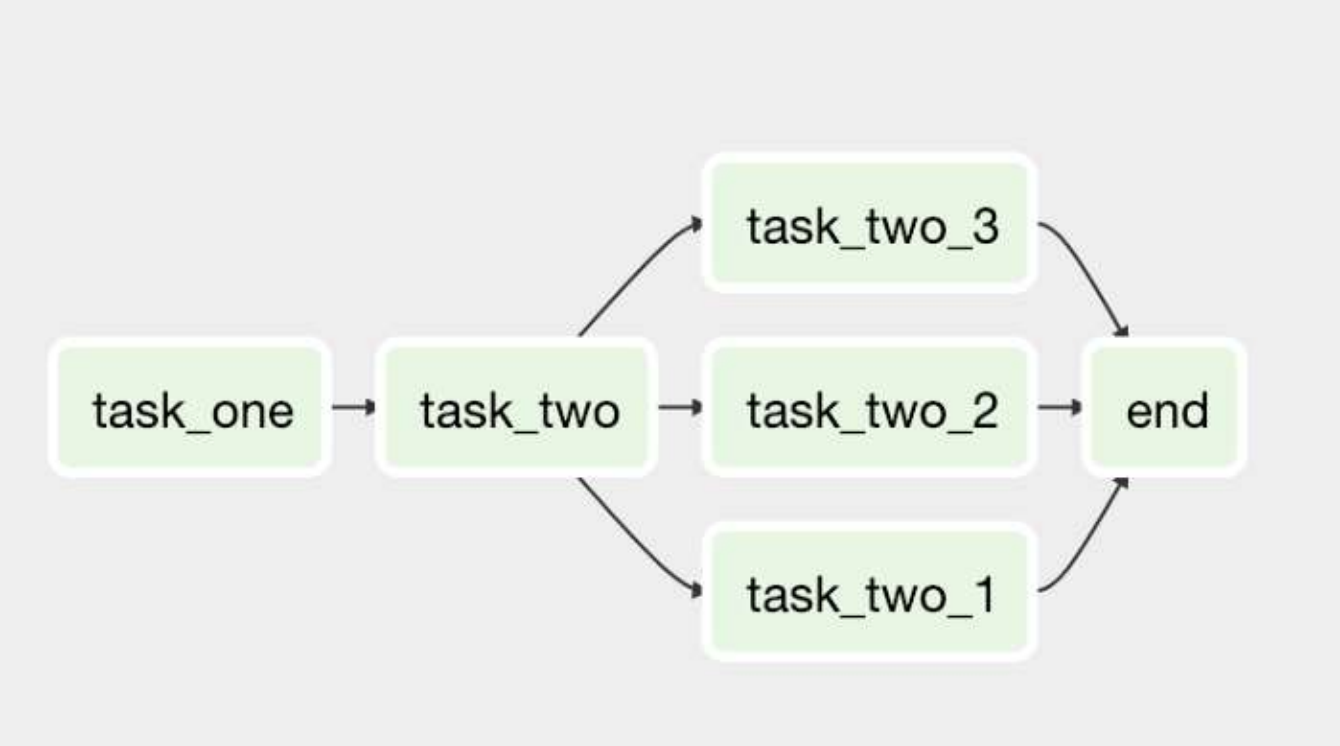
```python
# DAG with Context Manager

args = {
    'owner': 'airflow',
    'start_date': airflow.utils.dates.days_ago(2),
}

with DAG(dag_id='example_dag', default_args=args, schedule_interval='0 0 * * *') as dag:

        run_this_last = DummyOperator(
            task_id='run_this_last'
        )

        run_this_first = BashOperator(
            task_id='run_this_first',
            bash_command='echo 1'
        )

        run_this_first >> run_this_last
```

example_dag_with_context.py hosted with ♡ by GitHub view raw

The example (`example_dag.py` file) above just has 2 tasks, but if you have 10 or more then the redundancy becomes more evident. To avoid this you can use **Airflow** DAGs as **context managers** to automatically assign new operators to that DAG as shown in the above example (`example_dag_with_context.py`) using `with` statement.

(2) Using List to set Task dependencies

When you want to create the DAG similar to the one shown in the image below, you would have to repeat task names when setting task dependencies.




```
# Setting task dependencies (the NORMAL way)
task_one >> task_two
task_two >> task_two_1 >> end
task_two >> task_two_2 >> end
task_two >> task_two_3 >> end

# Using Lists (being a PRO :-D )
task_one >> task_two >> [task_two_1, task_two_2, task_two_3] >> end
```

airflow_list_task_dependencies.py hosted with ♡ by GitHub view raw

As shown in the above code snippet, using our normal way of setting task dependencies would mean that `task_two` and `end` are repeated 3 times. This can be replaced using

python lists to achieve the same result in a more elegant way.

(3) Use default arguments to avoid repeating arguments

Airflow allows passing a dictionary of parameters that would be available to all the task in that DAG.

For example, at DataReply, we use BigQuery for all our DataWareshouse related DAGs and instead of passing parameters like `labels`, `bigquery_conn_id` to each task, we simply pass it in `default_args` dictionary as shown in the DAG below.

```python
default_args = {
    'owner': 'airflow',
    'depends_on_past': False,
    'start_date': airflow.utils.dates.days_ago(2),
    # All the parameters below are BigQuery specific and will be available to all the tasks
    'bigquery_conn_id': 'gcp-bigquery-connection',
    'write_disposition': 'WRITE_EMPTY',
    'create_disposition': 'CREATE_IF_NEEDED',
    'labels': {'client': 'client-1'}
}

with DAG(dag_id='airflow_tutorial_gcp', default_args=default_args, schedule_interval=None) as d

    query_1 = BigQueryOperator(
        task_id='query_1',
        sql='select 1'
    )

    query_2 = BigQueryOperator(
        task_id='query_2',
        sql='select 1'
    )

    query_1 >> query_2
```

airflow_default_args.py hosted with ♡ by GitHub view raw

This is also useful when you want alerts on individual task failures instead of just DAG failures which I already mentioned in my last blog post on Integrating Slack Alerts in Airflow.

(4) The “params” argument

"params" is a dictionary of DAG level parameters that are made accessible in templates. These params can be overridden at the task level.

This is an extremely helpful argument and I have been personally using it a lot as it can be accessed in **templated** field with jinja templating using `params.param_name`. An example usage is as follows:

```python
# You can pass `params` dict to DAG object
default_args = {
    'owner': 'airflow',
    'depends_on_past': False,
    'start_date': airflow.utils.dates.days_ago(2),
}

dag = DAG(
    dag_id='airflow_tutorial_2',
    default_args=default_args,
    schedule_interval=None,
    params={
        "param1": "value1",
        "param2": "value2"
    }
)

bash = BashOperator(
    task_id='bash',
    bash_command='echo {{ params.param1 }}', # Output: value1
    dag=dag
)
```

airflow_params_usage_1.py hosted with ♡ by GitHub view raw

```python
# Passing `params` dict in `default_arg` dict
default_args = {
    'owner': 'airflow',
    'depends_on_past': False,
    'start_date': airflow.utils.dates.days_ago(2),
    'params': {
        "param1": "value2",
        "param2": "value1"
    }
}

dag = DAG(
    dag_id='airflow_tutorial_2',
    default_args=default_args,
    schedule_interval=None,
```

```
)

bash = BashOperator(
    task_id='bash',
    bash_command='echo {{ params.param1 }}', # Output: value2
    dag=dag
)
```

airflow_params_usage_2.py hosted with ♡ by GitHub view raw

```
# Passing `params` dict in tasks

default_args = {
    'owner': 'airflow',
    'depends_on_past': False,
    'start_date': airflow.utils.dates.days_ago(2),
}

dag = DAG(
    dag_id='airflow_tutorial_2',
    default_args=default_args,
    schedule_interval=None,
)

bash = BashOperator(
    task_id='bash',
    bash_command='echo {{ params.param1 }}', # Output: value3
    params={
        "param1": "value3",
        "param2": "value4"
    }
    dag=dag
)
```

airflow_params_usage_3.py hosted with ♡ by GitHub view raw

```
# You can override `params` dict passed in DAG object in `default_arg` dict
default_args = {
    'owner': 'airflow',
    'depends_on_past': False,
    'start_date': airflow.utils.dates.days_ago(2),
    'params': {
        "param1": "value2",
        "param2": "value2"
    }
}

dag = DAG(
    dag_id='airflow_tutorial_2',
    default_args=default_args,
```

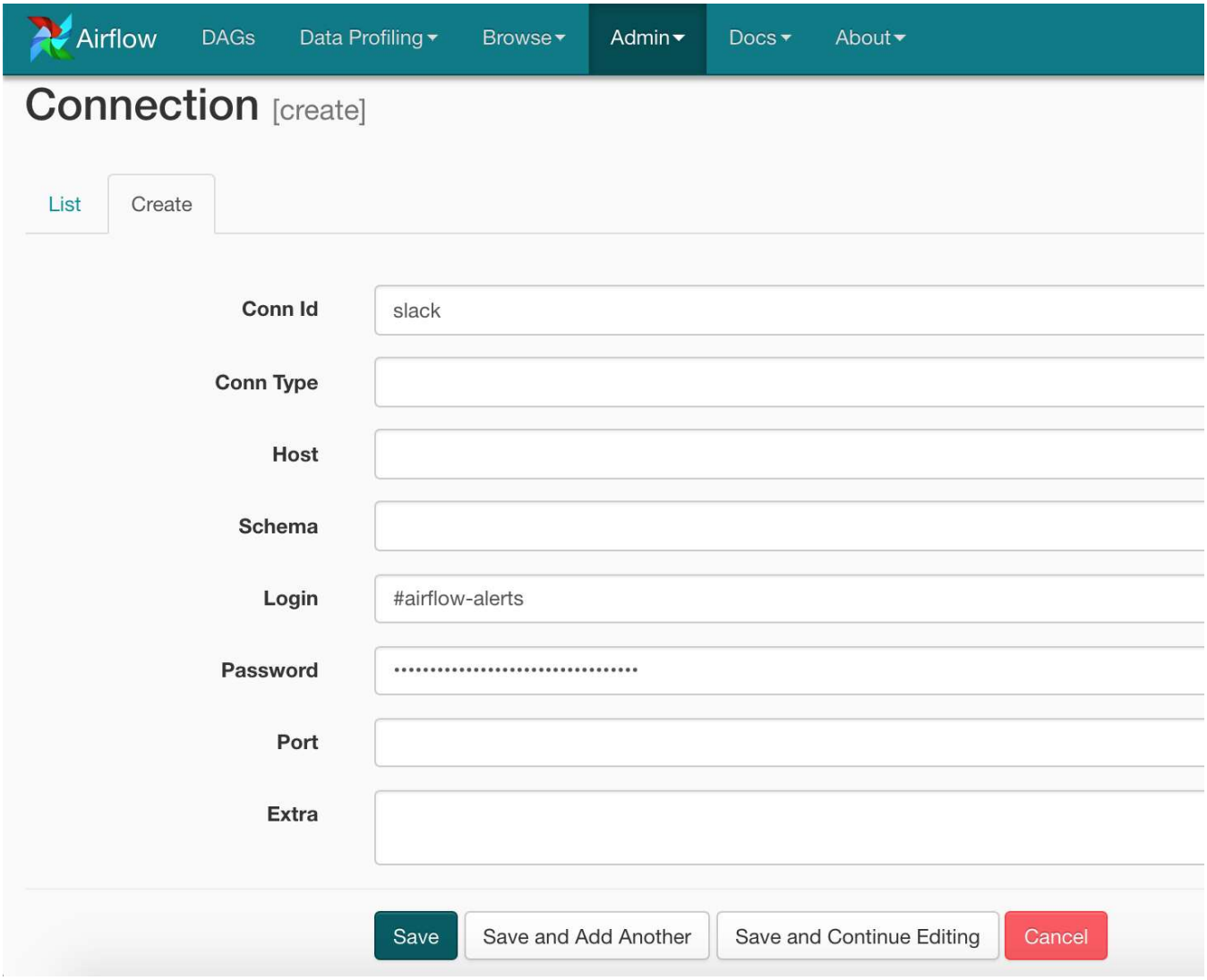
```
    schedule_interval=None,
    params={
        "param1": "value1",
        "param2": "value2"
    }
)

# You can override `params` dict passed in DAG object or `default_arg` in each individual tasks
bash = BashOperator(
    task_id='bash',
    bash_command='echo {{ params.param1 }}', # Output: value3
    params={
        "param1": "value3"
    }
    dag=dag
)
```

It makes it easy for you to write parameterized DAG instead of hard-coding values. Also as shown in the examples above `params` dictionary can be defined at 3 places: (1) In DAG object (2) In `default_args` dictionary (3) Each task.

(5) Storing Sensitive data in Connections

Most users are aware of this but I have still seen passwords stored in plain-text inside the DAG. For goodness sake — don't do that. You should write your DAGs in a way that you are confident enough to store your DAGs in a public repository.

By default, Airflow will save the passwords for the connection in plain text within the metadata database. The `crypto` package is highly recommended during Airflow installation and can be simply done by `pip install apache-airflow[crypto]`.



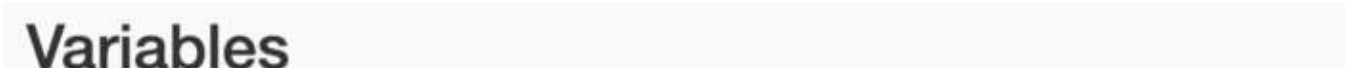
You can then easily access it as follows:

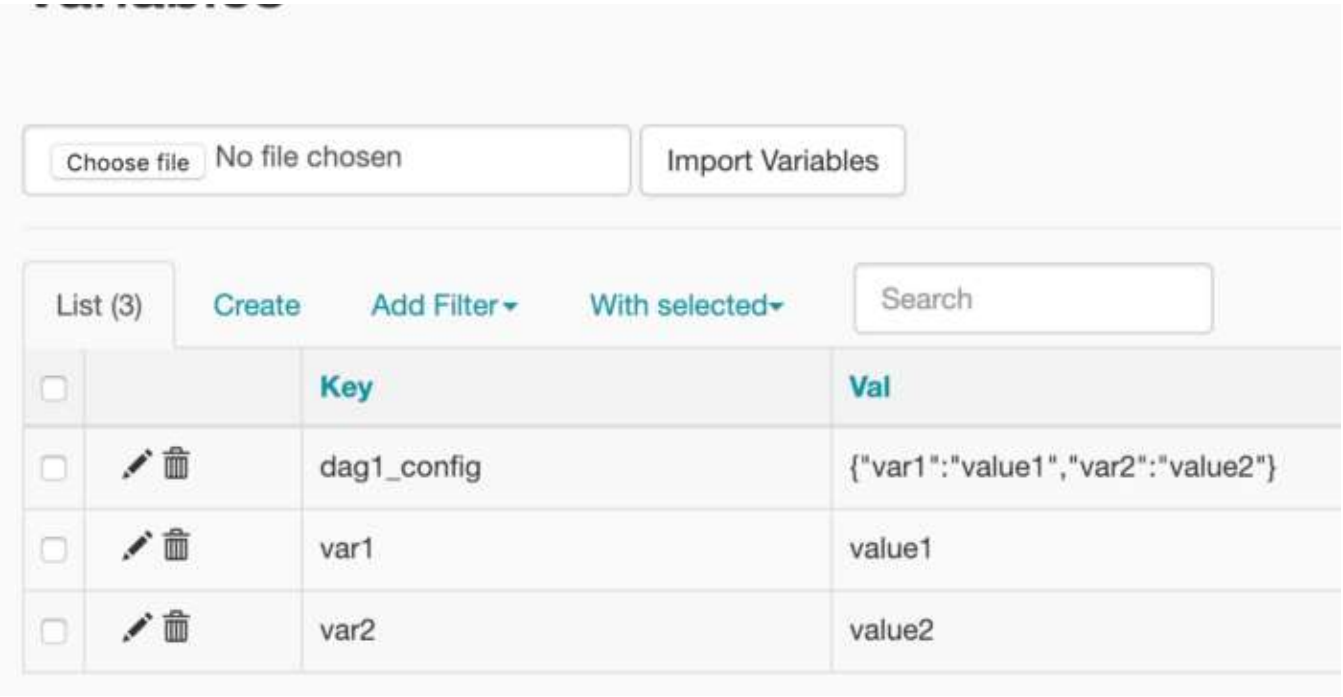
```
from airflow.hooks.base_hook import BaseHook
slack_token = BaseHook.get_connection('slack').password
```

(6) Restrict the number of Airflow variables in your DAG

Airflow Variables are stored in Metadata Database, so any call to variables would mean a connection to Metadata DB. Your DAG files are parsed every X seconds. Using a large number of variable in your DAG (and worse in `default_args`) may mean you might end up saturating the number of allowed connections to your database.

To avoid this situation, you can either just use a single Airflow variable with JSON value. As an Airflow variable can contain JSON value, you can store all your DAG configuration inside a single variable as shown in the image below:





As shown in this screenshot you can either store values in separate Airflow variables or under a single Airflow variable as a JSON field

You can then access them as shown below under **Recommended** way:

```python
from airflow.models import Variable

# Common (Not-so-nice way)
# 3 DB connections when the file is parsed
var1 = Variable.get("var1")
var2 = Variable.get("var2")
var3 = Variable.get("var3")

# Recommended Way
# Just 1 Database call
dag_config = Variable.get("dag1_config", deserialize_json=True)
dag_config["var1"]
dag_config["var2"]
dag_config["var3"]

# You can directly use it Templated arguments {{ var.json.my_var.path }}
bash_task = BashOperator(
    task_id="bash_task",
    bash_command='{{ var.json.dag1_config.var1 }} ',
    dag=dag,
)
```

airflow_json_variables.py hosted with ♡ by GitHub view raw

(7) The “context” dictionary

Users often forget the contents of the `context` dictionary when using `PythonOperator` with a callable function.

The context contains references to related objects to the task instance and is documented under the macros section of the API as they are also available to templated field.

```
{
        'dag': task.dag,
        'ds': ds,
        'next_ds': next_ds,
        'next_ds_nodash': next_ds_nodash,
        'prev_ds': prev_ds,
        'prev_ds_nodash': prev_ds_nodash,
        'ds_nodash': ds_nodash,
        'ts': ts,
        'ts_nodash': ts_nodash,
        'ts_nodash_with_tz': ts_nodash_with_tz,
        'yesterday_ds': yesterday_ds,
        'yesterday_ds_nodash': yesterday_ds_nodash,
        'tomorrow_ds': tomorrow_ds,
        'tomorrow_ds_nodash': tomorrow_ds_nodash,
        'END_DATE': ds,
        'end_date': ds,
        'dag_run': dag_run,
        'run_id': run_id,
        'execution_date': self.execution_date,
        'prev_execution_date': prev_execution_date,
        'next_execution_date': next_execution_date,
        'latest_date': ds,
        'macros': macros,
        'params': params,
        'tables': tables,
        'task': task,
        'task_instance': self,
        'ti': self,
        'task_instance_key_str': ti_key_str,
        'conf': configuration,
        'test_mode': self.test_mode,
        'var': {
            'value': VariableAccessor(),
            'json': VariableJsonAccessor()
        },
        'inlets': task.inlets,
        'outlets': task.outlets,
}
```

(8) Generating Dynamic Airflow Tasks

I have been answering many questions on StackOverflow on how to create dynamic tasks. The answer is simple, you just need to generate **unique** `task_id` for all of your tasks. Below are 2 examples on how to achieve that:

```
# Using DummyOperator
a = []
for i in range(0,10):
    a.append(DummyOperator(
        task_id='Component'+str(i),
        dag=dag))
    if i != 0:
        a[i-1] >> a[i]

# From a List
sample_list = ["val1", "val2", "val3"]
tasks_list = []
for index, value in enumerate(sample_list):
    tasks_list.append(DummyOperator(
        task_id='Component'+str(index),
        dag=dag))
    if index != 0:
        tasks_list[index-1] >> tasks_list[index]
```

airflow_dynamic_task.py hosted with ♡ by GitHub view raw

(9) Run “airflow upgradedb” instead of “airflow initdb”

Thanks to Ash Berlin for this tip in his talk in the First Apache Airflow London Meetup.

`airflow initdb` will create all default connections, charts etc that we might not use and don’t want in our production database. `airflow upgradedb` will instead just apply any missing migrations to the database table. (including creating missing tables etc.) It is also safe to run every time, it tracks which migrations have already been applied (using the Alembic module).

Let me know in the comments section below if you know something that would be worth adding in this blog post. Happy Airflow’ing :-)

Python Apache Airflow Airflow