Tutorial

This tutorial walks you through some of the fundamental Airflow concepts, objects, and their usage while writing your first pipeline.

Example Pipeline definition

Here is an example of a basic pipeline definition. Do not worry if this looks complicated, a line by line explanation follows below.

airflow/example_dags/tutorial.p view source

```
from datetime import timedelta

# The DAG object; we'll need this to
from airflow import DAG

# Operators; we need this to operate
from airflow.operators.bash import B
from airflow.utils.dates import days
# These args will get passed on to e
# You can override them on a per-tas
default_args = {
    'owner': 'airflow',
    'depends_on_past': False,
    'email': ['airflow@example.com']
    'email_on_failure': False,
    'email_on_retry': False,
    'retries': 1,
    'retry_delay': timedelta(minutes
    # 'queue': 'bash_queue',
    # 'pool': 'backfill',
    # 'priority_weight': 10,
    # 'end_date': datetime(2016, 1,
    # 'wait_for_downstream': False,
    # 'dag': dag,
    # 'sla': timedelta(hours=2),
    # 'execution_timeout': timedelta
    # 'on_failure_callback': some_fu
    # 'on_success_callback': some_ot
    # 'on_retry_callback': another_f
    # 'sla_miss_callback': yet_anoth
    # 'trigger_rule': 'all_success'
}
dag = DAG(
    'tutorial',
    default_args=default_args,
    description='A simple tutorial D
    schedule_interval=timedelta(days
    start_date=days_ago(2),
    tags=['example'],
)

# t1, t2 and t3 are examples of task
t1 = BashOperator(
    task_id='print_date',
    bash_command='date',
    dag=dag,
)

t2 = BashOperator(
    task_id='sleep',
    depends_on_past=False,
    bash_command='sleep 5',
```

```
    retries=3,
    dag=dag,
)
dag.doc_md = __doc__

t1.doc_md = """\
#### Task Documentation
You can document your task using the
`doc` (plain text), `doc_rst`, `doc_
rendered in the UI's Task Instance D
![img](http://montcs.bloomu.edu/~bob
"""
templated_command = """
{% for i in range(5) %}
    echo "{{ ds }}"
    echo "{{ macros.ds_add(ds, 7)}}"
    echo "{{ params.my_param }}"
{% endfor %}
"""

t3 = BashOperator(
    task_id='templated',
    depends_on_past=False,
    bash_command=templated_command,
    params={'my_param': 'Parameter I
    dag=dag,
)

t1 >> [t2, t3]
```

It's a DAG definition file

One thing to wrap your head around (it may not be very intuitive for everyone at first) is that this Airflow Python script is really just a configuration file specifying the DAG's structure as code. The actual tasks defined here will run in a different context from the context of this script. Different tasks run on different workers at different points in time, which means that this script cannot be used to cross communicate between tasks. Note that for this purpose we have a more advanced feature called XComs.

People sometimes think of the DAG definition file as a place where they can do some actual data processing - that is not the case at all! The script's purpose is to define a DAG object. It needs to evaluate quickly (seconds, not minutes) since the scheduler will execute it periodically to reflect the changes if any.

Importing Modules

An Airflow pipeline is just a Python script that happens to define an Airflow DAG object. Let's start by importing the libraries we will need.

airflow/example_dags/tutorial.p view source

```
from datetime import timedelta

# The DAG object; we'll need this to
from airflow import DAG

# Operators; we need this to operate
from airflow.operators.bash import B
from airflow.utils.dates import days
```

See Modules Management for details on how Python and Airflow manage modules.

Default Arguments

We're about to create a DAG and some tasks, and we have the choice to explicitly pass a set of arguments to each task's constructor (which would become redundant), or (better!) we can define a dictionary of default parameters that we can use when creating tasks.

airflow/example_dags/tutorial.p view source

```
# These args will get passed on to e
# You can override them on a per-tas
default_args = {
    'owner': 'airflow',
    'depends_on_past': False,
    'email': ['airflow@example.com']
    'email_on_failure': False,
    'email_on_retry': False,
    'retries': 1,
    'retry_delay': timedelta(minutes
    # 'queue': 'bash_queue',
    # 'pool': 'backfill',
    # 'priority_weight': 10,
    # 'end_date': datetime(2016, 1,
    # 'wait_for_downstream': False,
    # 'dag': dag,
    # 'sla': timedelta(hours=2),
    # 'execution_timeout': timedelta
    # 'on_failure_callback': some_fu
    # 'on_success_callback': some_ot
    # 'on_retry_callback': another_f
    # 'sla_miss_callback': yet_anoth
    # 'trigger_rule': 'all_success'
}
```

For more information about the BaseOperator's parameters and what they do, refer to the **`airflow.models.BaseOperator`** documentation.

Also, note that you could easily define different sets of arguments that would serve different purposes. An example of that would be to have different settings between a production and development environment.

Instantiate a DAG

We'll need a DAG object to nest our tasks into. Here we pass a string that defines the `dag_id`, which serves as a unique identifier for your DAG. We also pass the default argument dictionary that we just defined and define a `schedule_interval` of 1 day for the DAG.

airflow/example_dags/tutorial.p view source

```
dag = DAG(
    'tutorial',
    default_args=default_args,
    description='A simple tutorial D
    schedule_interval=timedelta(days
    start_date=days_ago(2),
    tags=['example'],
)
```

Tasks

Tasks are generated when instantiating operator objects. An object instantiated from an operator is called a task. The first argument `task_id` acts as a unique identifier for the task.

airflow/example_dags/tutorial.p view source

```
t1 = BashOperator(
    task_id='print_date',
    bash_command='date',
    dag=dag,
)

t2 = BashOperator(
    task_id='sleep',
    depends_on_past=False,
    bash_command='sleep 5',
    retries=3,
    dag=dag,
)
```

Notice how we pass a mix of operator specific arguments (`bash_command`) and an argument common to all operators (`retries`) inherited from BaseOperator to the operator's constructor. This is simpler than passing every argument for every constructor call. Also, notice that in the second task we override the `retries` parameter with `3` .

The precedence rules for a task are as follows:

1. Explicitly passed arguments
2. Values that exist in the `default_args` dictionary
3. The operator's default value, if one exists

A task must include or inherit the arguments `task_id` and `owner` , otherwise Airflow will raise an exception.

Templating with Jinja

Airflow leverages the power of Jinja Templating and provides the pipeline author with a set of built-in parameters and macros. Airflow also provides hooks for the pipeline author to define their own parameters, macros and templates.

This tutorial barely scratches the surface of what you can do with templating in Airflow, but the goal of this section is to let you know this feature exists, get you familiar with double curly brackets, and point to the most common template variable: `{{ ds }}` (today's "date stamp").

airflow/example_dags/tutorial.p view source

```
templated_command = """
{% for i in range(5) %}
    echo "{{ ds }}"
    echo "{{ macros.ds_add(ds, 7)}}"
    echo "{{ params.my_param }}"
{% endfor %}
"""

t3 = BashOperator(
    task_id='templated',
    depends_on_past=False,
    bash_command=templated_command,
    params={'my_param': 'Parameter I
    dag=dag,
)
```

Notice that the `templated_command` contains code logic in `{% %}` blocks, references parameters like `{{ ds }}` , calls a function as in `{{ macros.ds_add(ds, 7)}}` , and references a user-defined parameter in `{{ params.my_param }}` .

The `params` hook in `BaseOperator` allows you to pass a dictionary of parameters and/or objects to your templates. Please take the time to understand how the parameter `my_param` makes it through to the template.

Files can also be passed to the `bash_command` argument, like `bash_command='templated_command.sh'` , where the file location is relative to the directory containing the pipeline file (`tutorial.py` in this case). This may be desirable for many reasons, like separating your script's logic and pipeline code, allowing for proper code highlighting in files composed in different languages, and general flexibility in structuring pipelines. It is also possible to define your `template_searchpath` as pointing to any folder locations in the DAG constructor call.

Using that same DAG constructor call, it is possible to define `user_defined_macros` which allow you to specify your own variables. For example, passing

`dict(foo='bar')` to this argument allows you to use `{{ foo }}` in your templates. Moreover, specifying `user_defined_filters` allows you to register your own filters. For example, passing `dict(hello=lambda name: 'Hello %s' % name)` to this argument allows you to use `{{ 'world' | hello }}` in your templates. For more information regarding custom filters have a look at the Jinja Documentation.

For more information on the variables and macros that can be referenced in templates, make sure to read through the Macros reference.

Adding DAG and Tasks documentation

We can add documentation for DAG or each single task. DAG documentation only support markdown so far and task documentation support plain text, markdown, reStructuredText, json, yaml.

airflow/example_dags/tutorial.p view source

```
dag.doc_md = __doc__

t1.doc_md = """\
#### Task Documentation
You can document your task using the
`doc` (plain text), `doc_rst`, `doc_
rendered in the UI's Task Instance D
![img](http://montcs.bloomu.edu/~bob
"""
```

Setting up Dependencies

We have tasks `t1`, `t2` and `t3` that do not depend on each other. Here's a few ways you can define dependencies between them:

```
t1.set_downstream(t2)

# This means that t2 will depend on t1
# running successfully to run.
# It is equivalent to:
t2.set_upstream(t1)

# The bit shift operator can also be
# used to chain operations:
t1 >> t2

# And the upstream dependency with the
# bit shift operator:
t2 << t1

# Chaining multiple dependencies becomes
# concise with the bit shift operator:
t1 >> t2 >> t3

# A list of tasks can also be set as
# dependencies. These operations
# all have the same effect:
t1.set_downstream([t2, t3])
t1 >> [t2, t3]
[t2, t3] << t1
```

Note that when executing your script, Airflow will raise exceptions when it finds cycles in your DAG or when a dependency is referenced more than once.

Recap

Alright, so we have a pretty basic DAG. At this point your code should look something like this:

airflow/example_dags/tutorial.p view source

```
from datetime import timedelta

# The DAG object; we'll need this to
from airflow import DAG

# Operators; we need this to operate
from airflow.operators.bash import B
from airflow.utils.dates import days
# These args will get passed on to e
# You can override them on a per-tas
default_args = {
    'owner': 'airflow',
    'depends_on_past': False,
    'email': ['airflow@example.com']
    'email_on_failure': False,
    'email_on_retry': False,
    'retries': 1,
    'retry_delay': timedelta(minutes
    # 'queue': 'bash_queue',
    # 'pool': 'backfill',
    # 'priority_weight': 10,
    # 'end_date': datetime(2016, 1,
    # 'wait_for_downstream': False,
    # 'dag': dag,
    # 'sla': timedelta(hours=2),
    # 'execution_timeout': timedelta
    # 'on_failure_callback': some_fu
    # 'on_success_callback': some_ot
    # 'on_retry_callback': another_f
    # 'sla_miss_callback': yet_anoth
    # 'trigger_rule': 'all_success'
}
dag = DAG(
    'tutorial',
    default_args=default_args,
    description='A simple tutorial D
    schedule_interval=timedelta(days
    start_date=days_ago(2),
    tags=['example'],
)

# t1, t2 and t3 are examples of task
t1 = BashOperator(
    task_id='print_date',
    bash_command='date',
    dag=dag,
)

t2 = BashOperator(
    task_id='sleep',
    depends_on_past=False,
    bash_command='sleep 5',
    retries=3,
    dag=dag,
)
dag.doc_md = __doc__

t1.doc_md = """\
#### Task Documentation
You can document your task using the
`doc` (plain text), `doc_rst`, `doc_
rendered in the UI's Task Instance D
![img](http://montcs.bloomu.edu/~bob
"""
templated_command = """
{% for i in range(5) %}
    echo "{{ ds }}"
    echo "{{ macros.ds_add(ds, 7)}}"
    echo "{{ params.my_param }}"
{% endfor %}
"""

t3 = BashOperator(
```

```
t3 = BashOperator(
    task_id='templated',
    depends_on_past=False,
    bash_command=templated_command,
    params={'my_param': 'Parameter I
    dag=dag,
)

t1 >> [t2, t3]
```

Testing

Running the Script

Time to run some tests. First, let's make sure the pipeline is parsed successfully.

Let's assume we are saving the code from the previous step in `tutorial.py` in the DAGs folder referenced in your `airflow.cfg`. The default location for your DAGs is `~/airflow/dags`.

```
python ~/airflow/dags/tutorial.py
```

If the script does not raise an exception it means that you have not done anything horribly wrong, and that your Airflow environment is somewhat sound.

Command Line Metadata Validation

Let's run a few commands to validate this script further.

```
# initialize the database tables
airflow db init

# print the list of active DAGs
airflow dags list

# prints the list of tasks in the "tutorial" DAG
airflow tasks list tutorial

# prints the hierarchy of tasks in the "tutorial" DAG
airflow tasks list tutorial --tree
```

Testing

Let's test by running the actual task instances for a specific date. The date specified in this context is called `execution_date`. This is the *logical* date, which simulates the scheduler running your task or dag at a specific date and time, even though it *physically* will run now (or as soon as its dependencies are met).

```
# command layout: command subcommand dag_id task_id date

# testing print_date
airflow tasks test tutorial print_date 2015-06-01

# testing sleep
airflow tasks test tutorial sleep 2015-06-01
```

Now remember what we did with templating earlier? See how this template gets rendered and executed by running this command:

```
# testing templated
airflow tasks test tutorial templated 2015-06-01
```

This should result in displaying a verbose log of events and ultimately running your bash command and printing the result.

Note that the `airflow tasks test` command runs task instances locally, outputs their log to stdout (on screen), does not bother with dependencies, and does not communicate state (running, success, failed, …) to the database. It simply allows testing a single task instance.

The same applies to `airflow dags test [dag_id] [execution_date]`, but on a DAG level. It performs a single DAG run of the given DAG id. While it does take task dependencies into account, no state is registered in the database. It is convenient for locally testing a full run of your DAG, given that e.g. if one of your tasks expects data at some location, it is available.

Backfill

Everything looks like it's running fine so let's run a backfill. `backfill` will respect your dependencies, emit logs into files and talk to the database to record status. If you do have a webserver up, you will be able to track the progress. `airflow webserver` will start a web server if you are interested in tracking the progress visually as your backfill progresses.

Note that if you use `depends_on_past=True`, individual task instances will depend on the success of their previous task instance (that is, previous according to `execution_date`). Task instances with `execution_date==start_date` will disregard this dependency because there would be no past task instances created for them.

You may also want to consider `wait_for_downstream=True` when using `depends_on_past=True`. While `depends_on_past=True` causes a task instance to depend on the success of its previous task_instance, `wait_for_downstream=True` will cause a task instance to also wait for all task instances *immediately downstream* of the previous task instance to succeed.

The date range in this context is a `start_date` and optionally an `end_date`, which are used to populate the run schedule with task instances from this dag.

```
# optional, start a web server in debug mode in the background
# airflow webserver --debug &

# start your backfill on a date range
airflow dags backfill tutorial \
    --start-date 2015-06-01 \
    --end-date 2015-06-07
```

What's Next?

That's it, you have written, tested and backfilled your very first Airflow pipeline. Merging your code into a code repository that has a master scheduler running against it should get it to get triggered and run every day.

Here's a few things you might want to do next:

See also

- Read the Concepts page for detailed explanation of Airflow concepts such as DAGs, Tasks, Operators, etc.
- Take an in-depth tour of the UI - click all the things!
- Keep reading the docs!
  - Review the how-to guides, which include a guide to writing your own operator
  - Review the Command Line Interface Reference
  - Review the List of operators
  - Review the Macros reference
- Write your first pipeline!

Previous Next

Want to be a part of Apache Airflow? Join community

License Donate Thanks
Security