Airflow Training

How to use this document

The purpose of this document is to show in what cases Apache Airflow can be used, how should it be done. Please keep in mind that I'm will show here core concepts used by airflow and that should be a base for resolving your uses cases.

All code that is provided in this document can be found in GitHub repository: https://github.com/Szczensny/teaching_airflow. It is also place where you can download the pre-configured local Airflow environment to try out everything that you will learn here.

In this document I will use Apache Airflow in version 1.10.1, but if you want to setup your own cluster – please use the newest possible stable version.

Please be also aware that using airflow required to have basic knowledge of Python language and SQL. If you are not familiar with it – I suggest to find some free materials on the internet i.e.

- *https://www.coursera.org/specializations/python*
- *https://www.codecademy.com/learn/learn-sql#*

Convention for code

```
In gray boxes you will find code that is inside repository with this file
```

```
In green boxes is code that was is not included in the repository
```

Parameters are served as italic font.

What is Apache Airflow

“Airflow is a platform to programmatically author, schedule and monitor workflows. When workflows are defined as code, they become more maintainable, versionable, testable, and collaborative.

Airflow is not a data streaming solution. Tasks do not move data from one to the other (though tasks can exchange metadata!). Airflow is not in the Spark Streaming or Storm space, it is more comparable to Oozie or Azkaban.
Workflows are expected to be mostly static or slowly changing. You can think of the structure of the tasks in your workflow as slightly more dynamic than a database structure would be. Airflow workflows are expected to look similar from a run to the next, this allows for clarity around unit of work and continuity.” [1]

Apache Airflow is an open source platform that’s allows users to easy create and execute data related processes in ETL[2] *approach using Python code. Processes are called workflows (or pipelines – those terms are equal to each other) that are organized in DAG (Directed Acyclic Graphs)*[3] *. It was started October 2014 Maxime Beauchemin and till today is still develop by Airflow team and many individual developers from all around the world.*

[1] https://airflow.apache.org/docs/stable/
[2] https://en.wikipedia.org/wiki/Extract,_transform,_load
[3] https://en.wikipedia.org/wiki/Directed_acyclic_graph

Airflow Architecture

Airflow as a platform for creating and executing workflows is using servial components to make it happened. Let's go through them one by one.

Webserver
Webserver is the place when users using the web browser are able to see current dags that are loaded into Airflow as well with their statuses. Here you are also able to manually run dag or see logs from it. Another functionality is make changes in Airflow objects - Connections, Variable, Users - but that depends on the privileges that are assigned to your profile. Information about dags is downloaded from backend database and in the same time from parsed python files that are on the instance (In lasted version of Airflow parsing dag on webserver is dropped and replaced by dag serialisation concept[4])

Scheduler
This component have two main reasons:
Parse python dag files, then save metadata about dag into backend database (table dag) - interval for this is set by dag_dir_list_interva variable in Airflow configuration file.
Second reason is to check in schedules passed into dags and if the expected time for dag run is hit - send this information to message queue.

Executor
This service is responsible for

- *Getting trigger signal from scheduler to run particular code to execution.*
- *Distributed work to worker nodes if type of executor is allowing for that*
- *Get back information about status of the tasks assigned to worker*
- *Upload this information to backend database*

In our case we are using CeleryExecutor that allows to distribute tasks to many workers (don't have to be on the same machine) using message queue as medium of exchange information between Celery[5] and workers.

Worker
This is the place where in the end your pipeline code is executed. Workers are pulling data about what should be executed from the medium used by Executor, execute the code and then return the status of it back to the executor.
Every worker have also assigned queue name that should be checked for executing new dags, thats allows to redirect execution to only one group of workers or individual instance if its needed in example for security reasons.

Message queue
This component is required by CeleryExecutor. It allows to exchange information between celery and workers instances using message service (Redis, RabbitMQ, Amazon SQS)

---

[4] https://airflow.apache.org/docs/stable/dag-serialization.html
[5] https://docs.celeryproject.org/en/stable/getting-started/introduction.html

Database
This is the very last component in use. Database is here - obvious sherlock - to store information. To be more precise - Informations about:

- *Connections (in current version can be get from secret backend)*
- *Users (in current version can be get from secret backend)*
- *Dags metadata (dag runs, dag place of storage dag etc.)*
- *Tasks logs (where executed etc.)*
- *Tasks instances*
- *Variables*
- *xComs*

In our case we are using Postgres as backend database, but it is possible to use any engine that is compatible with python SqlAlchemy libliary[6], but it is recommended to use Postgres or MySql database as backend.

Core Airflow concepts

Task
It is a logicali encapsulated part of python code that is a single step in the pipeline. Tasks should be focused to resolve part of the problem that you coded into pipeline.

Task instance
This is a tasks that is/was executed in some point in time and have a context of the execution. More about context later on.

DAG
Directed Acyclic Graph of the process written in pipeline. It is representing all tasks that are assigned to it and dependencies between them.
*Very important to remember - in pipeline each tasks can be executed **only once**. If you need to do the same thing two times or more - you need to declare each tasks for each execution needed.*

Hook
*Component that is responsible for **directly interacting** with data source for example with database, SFTP etc.*
Hooks can be divided into Airflow hooks (made by Airflow developers), Contrib hoks (made by community, but added to Airflow repository) and Custom hooks (made by you, but not pushed to Airflow repo).

[6] https://docs.sqlalchemy.org/en/13/core/engines.html

Operator
Is a component that is ***encapsulated part of python code with logic of data processing****.*
They can use multiple hooks to interact with data sources. ***Instance of Operator is an task in the pipeline.***
Operators can be divided into Airflow operators (made by Airflow developers), Contrib operators (made by community, but added to Airflow repository) and Custom operators (made by you, but not pushed to Airflow repo).

Sensor
Is a component that is using hooks to ***detect some action on the data source*** *- For example if file was loaded into external service, new row added in table etc.*
Sensors can be divided into Airflow sensors (made by Airflow developers), Contrib sensors (made by community, but added to Airflow repository) and Custom sensors (made by you, but not pushed to Airflow repo).

Variable
Is an Airflow object that is storing some information that can be reused in any Airflow object. The value of variable is saved in backend database.
Important!
As Variable is saved in database, so don't store big amount of data in it! If you need to save big amount of information (i.e. content of 200mb file) consider loading it from external storage like Amazon S3)

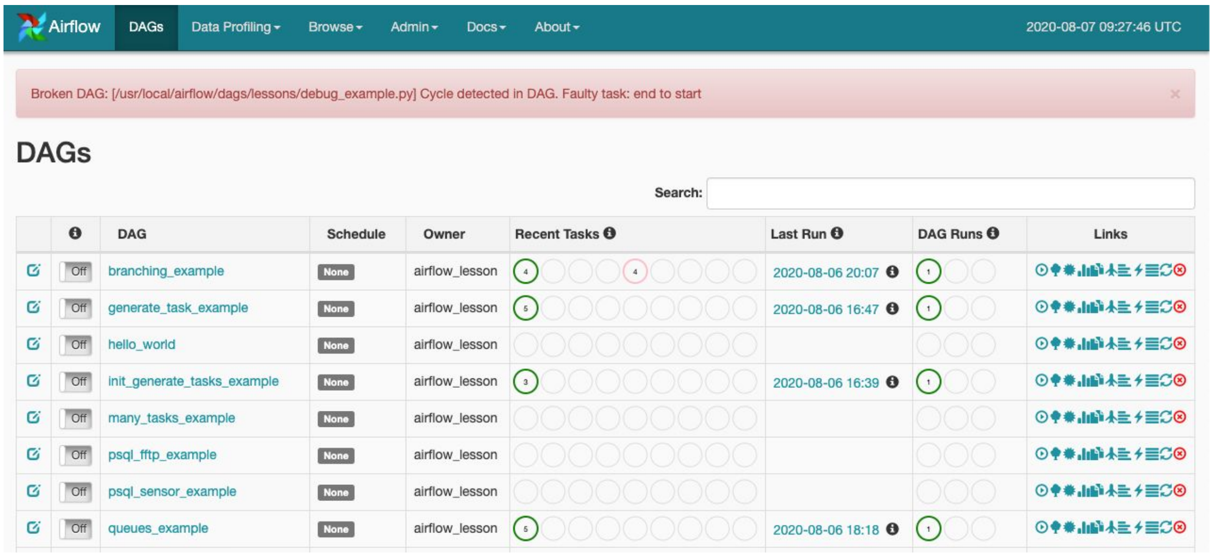
xCom
Object in airflow that is able to store information and share it between tasks. It is also saved in backend database, so if you need to pass big amount information between tasks - consider putting them in external file storage like Amazon S3 and pass just the reference to it via xCom

Pool
Is an object that holds information how many of your task instances can be runned at the same time in paralel. To use it - you have to pass name of pool to your task. If not provided Airflow will try to execute all parallel tasks in the same time.

Context:
Is a set of information assigned to your task instance when its running. It contains for example date of execution, status of each tasks, xcoms etc.

Airflow User Interface

After correct setup of your local environment, you can go to your web browser and open http://localhost:8080 - you should see interface similar to this:



Broken DAG: [/usr/local/airflow/dags/lessons/debug_example.py] Cycle detected in DAG. Faulty task: end to start

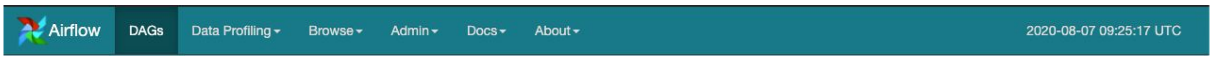
DAGs

Search:

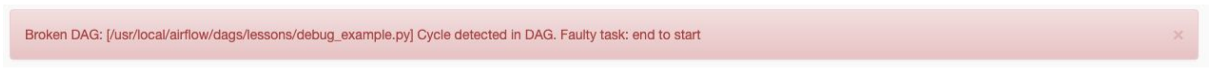
		DAG	Schedule	Owner	Recent Tasks	Last Run	DAG Runs	Links
	Off	branching_example	None	airflow_lesson	4 4	2020-08-06 20:07	1	
	Off	generate_task_example	None	airflow_lesson	5	2020-08-06 16:47	1	
	Off	hello_world	None	airflow_lesson				
	Off	init_generate_tasks_example	None	airflow_lesson	3	2020-08-06 16:39	1	
	Off	many_tasks_example	None	airflow_lesson				
	Off	psql_fftp_example	None	airflow_lesson				
	Off	psql_sensor_example	None	airflow_lesson				
	Off	queues_example	None	airflow_lesson	5	2020-08-06 18:18	1	

Important!
Depending on cluster configuration this view can be different, because it depends on user privileges and general cluster settings (some parts could be hidden). In our case we have access to all options as Admin user.

Lets see what do we have here.



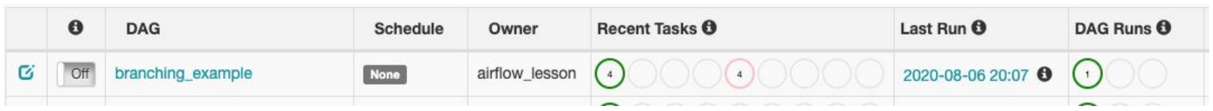
First part is the navigation panel, that allows you to access general functionality of Airflow like for example, user creation and status, connections settings, cluster configuration etc.



Broken DAG: [/usr/local/airflow/dags/lessons/debug_example.py] Cycle detected in DAG. Faulty task: end to start

Next part is error information if any of pipeline file is corrupted. We will talk how to deal with this kind of situation later on.

On the right site you have search box, where you can search dags by its name. It's very useful when you have a lot of dags in your cluster.

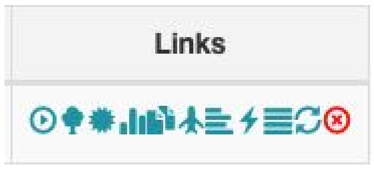


		DAG	Schedule	Owner	Recent Tasks	Last Run	DAG Runs
	Off	branching_example	None	airflow_lesson	4 4	2020-08-06 20:07	1

Next part is list of dags that out of the box give you some value information. Starting from left site

- *Edit button - allows to change some basic information about dag like for example its owner.*
- *On/Off button - Allows to turn on/off pipeline for executing.*
- *Schedule - None or cron format schedule time when pipeline code should be executed.*
- *Owner - person/team that created/mainten pipeline. If not provided by pipeline use the cluster setting.*
- *Recent Tasks - Status of tasks in pipeline from current run (if executed in the time of looking on it) or from last pipeline execution.*
- *Last run - Time stamp when last time pipeline was executed*
- *Dag Runs - Summary of all execution of pipeline in cluster. green = success, light green = running, red = failed*

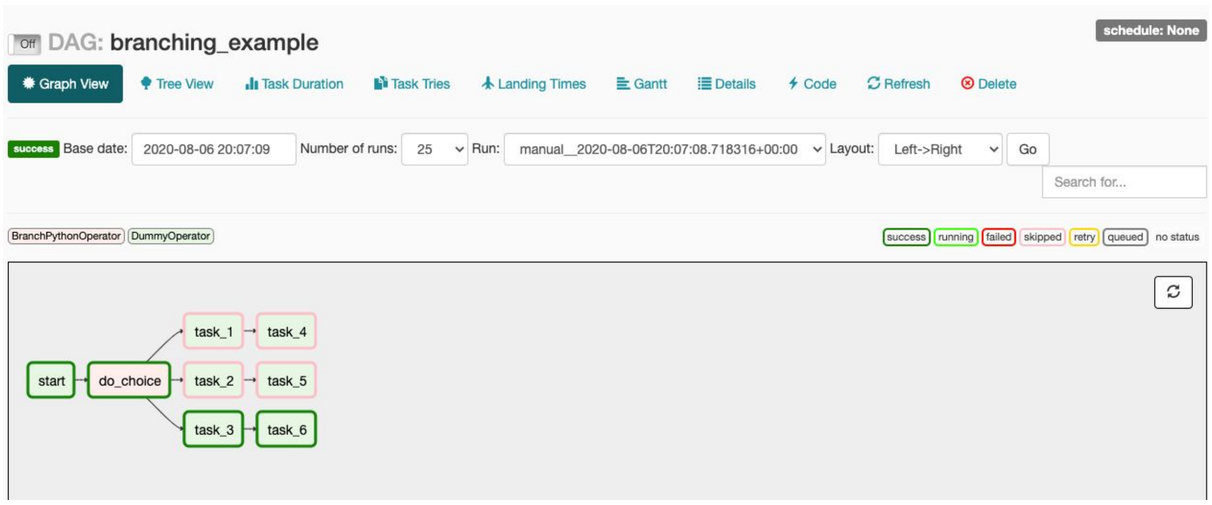
Last column in dag list are actions:



And again from left:

- *Trigger Dag - Allows to run dag manually without waiting for execution schedule. Will work only if dag is turned on.*
- *from second to seven - Different views with dag runs statistic. try it out!*
- *Code view - Look at code pipeline in web browser (just read, no edit there)*
- *Logs - List of logs from pipeline*
- *Refresh - Force Airflow to reload code from pipeline*
- *Delete - remove dag from cluster (metadata and files)*

When you click on dag name from the table you will be moved to another view of current run/ last run:



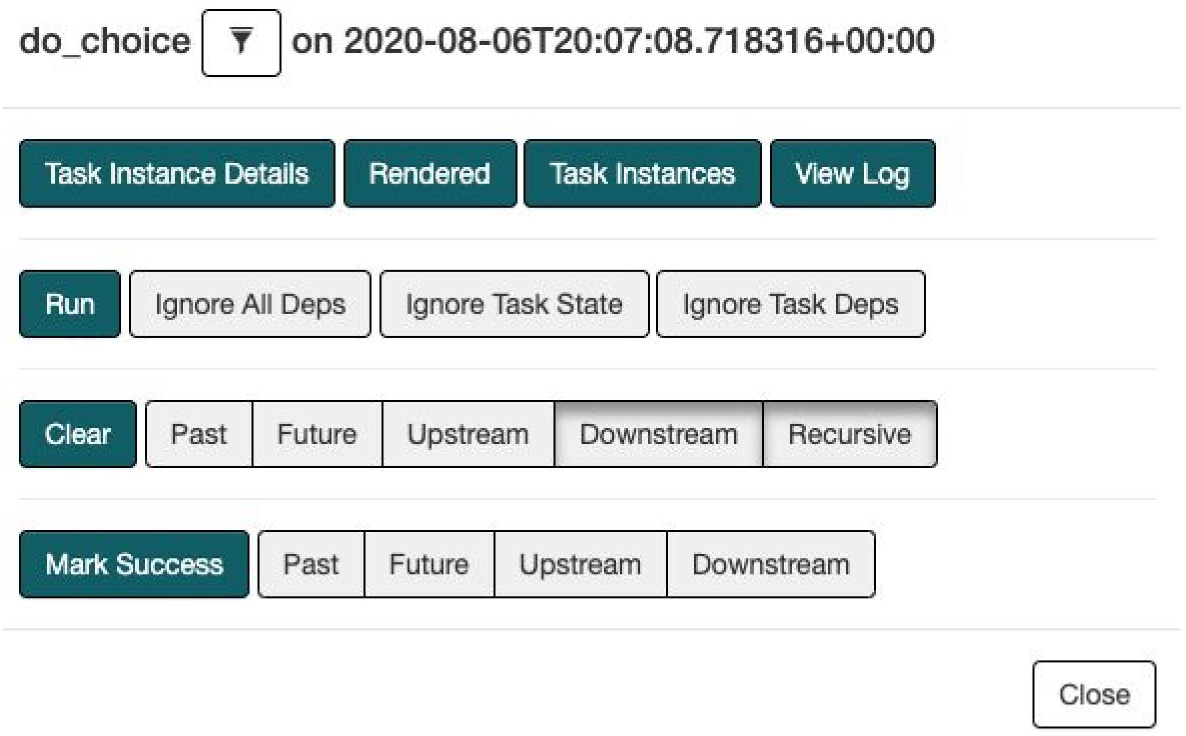
success Base date: 2020-08-06 20:07:09 Number of runs: 25 Run: manual__2020-08-06T20:07:08.718316+00:00 Layout: Left->Right

BranchPythonOperator DummyOperator

Now from the top:

- *The same schedule as from main view*
- *On/Off button*
- *The same list of views with statistic, refresh button and delete (trigger button is only in main view)*
- *Place where to which dag run you want to see diagram*
- *List of operators used in pipeline and colour legend for diagram*
- *Graph of your pipeline*

Another options can be shown when you left click on task in your pipeline.



And last time from the top:

- *Name of task and timestamp of execution*
- *task instance details - list of parameters that ware in context of your pipeline run (even more this task run)*
- *Rendered - Shows all templates that were templated in task run.*
- *Task instances - list of task instance details that were generated for task in cluster*
- *View Log - View logs that were generated during task execution.*
- *Run - Manually run in pipeline*
- *Clear - remove metadata information of task statuses*
- *Mark Success - update task instance metadata with success status.*

How to write a pipeline

Folder structure

In order to let Airflow see your pipeline python file have to be putted in the dag folder that is declared in the configuration file under the dags_folder. It is a good practice to put them in separate folders if you are able to group them for example by topic. In our case you should put your file under dags folder in cloned repository.

Overview of example dag file:

Below you can see an example of dag file. In cloned repository it can be found in file dags/lessons/hello_world.py. Let’s go through it step by step and explain each part of it.

```
"""
This is test pipeline to see if airflow is
working properly

Data_in: None
Data_out: None
Depend_on: None
@author: Rafal Chmielewski
@team: Airflow Learning
@stakeholders: People who learns
"""
from airflow.operators.python_operator
import PythonOperator
from airflow.operators.dummy_operator
import DummyOperator
from airflow.models import DAG
import datetime
import logging


def say_hello(**context):
    """
    Function is puting example string into
task log.
    :param context:
    :return:
    """
    logging.info(f'Everything Works!
{datetime.datetime.now()}')

dag = DAG(
    dag_id='hello_world',
    schedule_interval=None,
    start_date=datetime.datetime(2020, 1, 1),
    default_args={"owner": "airflow_lesson"}
)

start = DummyOperator(
    task_id='start_dag',
    dag=dag
)

hello = PythonOperator(
    task_id='say_hello',
    python_callable=say_hello,
    dag=dag
)

end = DummyOperator(
    task_id='end_dag',
    dag=dag
)

start >> hello >> end
```

Litening of hello_world.py

```
"""
This is test pipeline to see if airflow is working properly

Data_in: None
Data_out: None
Depend_on: None
@author: Rafal Chmielewski
@team: Airflow Learning
@stakeholders: People who learns
"""
```

First position here is a docstring that describes the purpose of dag, datasets that is operating on, information about author and benefitiens of the process. It is not required to add it, but it is a very good practice.

```
from airflow.operators.python_operator import PythonOperator
from airflow.operators.dummy_operator import DummyOperator
from airflow.models import DAG
import datetime
import logging
```

Next part are imports of all libraries or objects that are need in your pipeline.
In case of importing plugins from delivered from Airflow you need to always start with airflow key, but in case of custom plugins, just use a name of the plugin from plugins folder. If custom plugin is in directory - then go with formula <folder_name>.<another_folder>.<file_name>. I will show example of that later on.

```
def say_hello(**context):
    """
    Function is puting example string into task log.
    :param context:
    :return:
    """
    logging.info(f'Everything Works! {datetime.datetime.now()}')
```

In every dag you can (but don’t have to) declare as much extra python functions as needed. They can be called by task, or by another function. In case of using it directly by task - don’t declare any required arguments, only kwargs (usually in Airflow code named as context).

```
dag = DAG(
    dag_id='hello_world',
    schedule_interval=None,
    start_date=datetime.datetime(2020, 1, 1),
)
```

Here is the main point of declaration of pipeline and all required information for Airflow to process it.

dag_id *is the name of your pipeline that will be visible in the web server UI. Name should be unique across the all pipelines.*

schedule_interval *- time when pipeline will be automatically executed by Airflow set in crontab syntax[7]. In case of value None - it will be processed, by Airflow but not executed. To do so you need to trigger it by Airflow CLI or webserver UI.*

start_date *- datetime object (with time aspect), date from which scheduling auto execution should started. For example if schedule_interval is set to 0 */1 * * * every hour, current date is 2020-07-20 and start date is set to 2020-08-01 18:30:00 - the first execution will happen at 2020-08-01 at 19:00.*

One of additional parameter not mentioned in listening is default_args which as an argument get a dictionary of parameters that will be passed to tasks, that are belonging (are assigned to dag). Usually you put there parameters that are repeating in tasks.

Of course more parameters can be added here like end_date, max_active_runs - you can find all of them in [Dag class code](#).

Technically in one python file you can declare more than one dag, but it will most likely it will be hard to easy read code in file or debug in case of any problem. Good practices is to put 1 dag in one python file.

[7] https://man7.org/linux/man-pages/man5/crontab.5.html

```
start = DummyOperator(
    task_id='start_dag',
    dag=dag
)

hello = PythonOperator(
    task_id='say_hello',
    python_callable=say_hello,
    provide_context=True,
    dag=dag
)

end = DummyOperator(
    task_id='end_dag',
    dag=dag
)
```

Next part is declaration of each tasks that will be executed in pipeline. Each variable is an instance of Operator or Sensor class that you want to use. The common attributes that are passed to them are:

task_id *- name of your task that will be visible in web server UI. It have to be unique in all pipeline!*
dag *- assignment to dag - be careful, because it can be also done with context manager approach (with Dag(parameters) as Dag:), but there are corner cases when it is wrongly interpreted by Airflow.*

Rest of arguments are different between operators and sensors. To use them please check the source code of it, where at the very beginning is docstring with all parameters listed and they explanation. Let's take a closer look at it in PythonOperator[8]*.*

[8] https://github.com/apache/airflow/blob/76a5fc4d2eb3c214ca25406f03b4a0c5d7250f71/airflow/operators/python_operator.py#L38

```
"""
    Executes a Python callable
    :param python_callable: A reference to an object that is callable
    :type python_callable: python callable
    :param op_kwargs: a dictionary of keyword arguments that will get unpacked
        in your function
    :type op_kwargs: dict
    :param op_args: a list of positional arguments that will get unpacked when
        calling your callable
    :type op_args: list
    :param provide_context: if set to true, Airflow will pass a set of
        keyword arguments that can be used in your function. This set of
        kwargs correspond exactly to what you can use in your jinja
        templates. For this to work, you need to define `**kwargs` in your
        function header.
    :type provide_context: bool
    :param templates_dict: a dictionary where the values are templates that
        will get templated by the Airflow engine sometime between
        ``__init__`` and ``execute`` takes place and are made available
        in your callable's context after the template has been applied. (templated)
    :type templates_dict: dict of str
    :param templates_exts: a list of file extensions to resolve while
        processing templated fields, for examples ``['.sql', '.hql']``
    :type templates_exts: list(str)
    """
```

As you see the authors of PythonOperator decide to allow arguments:

- *python_callable - which is pointing to python function that you want to execute, but please remember that its need to be passed* ***without parenthesis.*** *In other case airflow can throw an error about syntax!*
- *op_kwargs - dictionary that will be translated as class attributes*
- *op_args - list of positional arguments [order the same as in class definition]*
- *provide_context - Bool value if task instance information should be passed to kwargs*
- *templates_dict - dictionary where each key can be used in templates passed to operator*
- *templates_ext - list of file extensions that should be templated.*

In nexts chapters we gonna get a closer look on templating.

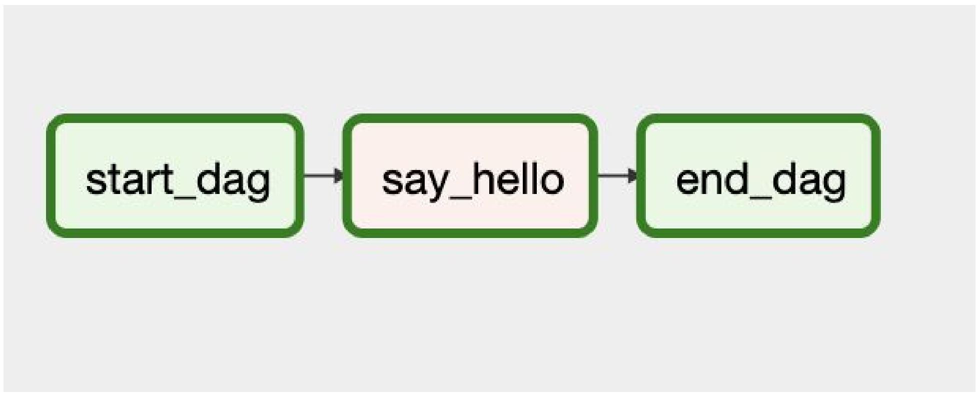
But there is even more of possibilities, because PythonOperator is a child from BaseOperator for example on_success_callback which after sucesfull execution of task will execute function provided.

There are many possibilities, so it is always good if any of parent class can offer you something that you need to do.

```
start >> hello >> end
```

The very last step in pipeline is to determine how tasks inside are related to each other. In our example we say that first should run task start, then hello and as last end.
By default in this case each of them will be executed if all previous tasks was executed with successful status, of course you can manipulate those by passing trigger_rule[9] parameter into your task declaration.

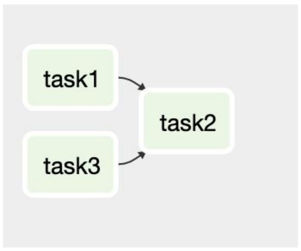
In web server UI it will look like this:



There are few ways how to describe dependencies. The first one is using python bit operator '>>' or '<<'. Let's look on example:

```
task1 >> task2 << task3
```

You may ask how to interpret that? Here we say that task1 should be executed before task2 by task1 >> task2, but also we say task2 << task3 which can be written also as task2 >> task3. On the graph it will look like this:



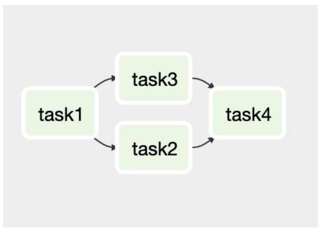
So tasks1 and task3 are on the same 'level of execution', which means that by default Airflow will execute them on the same time.

```
task1.set_downstream(task2)
task2.set_upstream(task3)
```

[9] https://airflow.apache.org/docs/1.10.1/concepts.html#trigger-rules

The same result we can achieve, by using set_upstream and set_downstream methods on our tasks. Both of them accept as argument task object to which it should be related.

But what in case when you need make a situation like this:



According to previous examples the solution should be:

```
task1 >> task2
task1 >> task3
task2 >> task4
task3 >> task4
```

But that is quite many lines for such small pipeline. Imagine that in the middle you don't want to put 2 task, but 100. To make life easier you can put between task1 and task4 list of tasks. They will be executed by airflow in the same time (sill can be limited by pool parameter).
In our example it will look like:

```
task1 >> [task2, task3] >> task4
```

And let's agree that this syntax is better to be read by human.

Important thing to remember.

Web server and Scheduler components of Airflow is scanning files in order to find new pipelines and changes in already registered in interval. That's why all code that is making calculations on data or interacting with data sources should be wrapped in functions. If not Airflow will try to execute this code with every scan made, which can lead to slow down or crush of Airflow cluster. Also if you want to put some information to logs never use print() meethod, but instead use logging library [logging.info(), logging.warn(), logging.().error, logging.debug()]. That's very important if logs from cluster are used in external alering tool like for example Sentry[10] that can just skip messages that were used print method.
Homework:

1. *Write a pipeline that write to the logs a dictionary with your name, company email address, favorite colour.*
2. *Check Airflow documentation[11] on what kind of operators/hooks are in Airflow. Which of them you think are the most common one in use for us?*

[10] https://sentry.io/
[11] https://airflow.apache.org/docs/1.10.1/code.html#operators

More than basic pipeline

Initial pipeline

So far we created a pipeline where we hardcoded many information, but what if we need to change one small parameter and we can't wait for Continuous delivery process? Or we want have possibility to use different settings in different environments? Or we just want to be sure that all information that needs to be generated for pipeline is in place? Answer is simple - the init pipeline.

Technically speaking its you can write it in similar way that we have the hello_wold example. The same rules apply here, but the purpose is different. Let's say that we have operator that needs to do some operation on all tables in one schema in our database and we want to generate this code. As we learned earlier airflow have to know exactly how many tasks is in pipeline, before it let to execute it.

To make this kind of declaration for Airflow we gonna use a initialization pipeline.

```
"""
This pipeline initialise base data for generate_tasks_example.py .

Data_in:
   Postgres:
      - other schema
Data_out: None
Depend_on: None
@author: Rafal Chmielewski
@team: Airflow Learning
@stakeholders: People who learns
"""
from airflow.operators.dummy_operator import DummyOperator
from airflow.operators.python_operator import PythonOperator
from airflow.hooks.postgres_hook import PostgresHook
from airflow.models import DAG, Variable, Pool, datetime
from airflow.settings import Session
import logging
import pandas as pd
```

At beginning of our pipeline we put as always our docstring with description and all needed imports.

```
def generate_init():
    """
    Function is checking names of tables in postgres other.
    Then information is combined with standard params and pushed as Airflow Variable
object.
    Also pool is created.
    :return:
    """
```

First part of our code will be a function that will generate all information needed for our main pipeline. In this case we want to get a list of all tables in schema order in postgres database. Here we gonna use PostgresHook to retrieve those names and put it into Python list.

```
    psql_hook = PostgresHook('airflow_docker_db')
    eng = psql_hook.get_sqlalchemy_engine()
    df = pd.read_sql("select table_name from information_schema.tables where
table_schema='other';",
                     con=eng)
    table_list = df['table_name'].tolist()
```

Next we don’t want to do too much connection at the same time to the database. To make sure that only few task run at the same time we will declare pool.

```
    try:
        pool = Pool()
        pool.slots = 1
        pool.description = 'How many tasks can run at once'
        pool.pool = 'generate_tasks'
        session = Session()
        session.add(pool)
        session.commit()
    except Exception as ex:
        logging.info(Could not set pool. Details: {ex}')
```

When we have a pool lets combine all information that will be used by pipeline into dictionary.

```
    init_data= {
        'psql_conn_id': 'airflow_docker_db',
        'table_list': table_list,
        'pool': 'generate_tasks'
    }
try:
    Variable.set(key='generate_tasks', value=init_data, serialize_json=True)
except Exception as ex:
    logging.info(f'Could not set global variable. Details: {ex}')
```

As you see in Variable object we define key for it which is a reference name in Airflow for dictionary that we provided.
Keep in mind that all variables values ***are stored in backend database of airflow****, so don't put there to many information. If you need to do so - think how to use it from different place like S3 bucket.*

```
dag = DAG(
    dag_id='init_generate_tasks_example',
    schedule_interval=None,
    start_date=datetime(2020, 1, 1),
    default_args={"owner": "airflow_lesson"}
)
start = DummyOperator(
    task_id='start',
    dag=dag
)

init = PythonOperator(
    task_id='init_pipeline',
    python_callable=generate_init,
    dag=dag
)

end = DummyOperator(
    task_id='end',
    dag=dag
)

start >> init >> end
```

Last part of our pipeline is standard dag and tasks declaration. Our function will be run by PythonOperator. Keep in mind that init pipeline suppose to by run manually whenever is need for that, so schedule interval here should be put to None value.

Using Variable and Pool in pipeline and generate list of task.

We gonna start with our docstring and all needed imports

```
"""
This pipeline shows how generate task based on variable.

Data_in:
    Postgres:
        - other schema
Data_out: None
Depend_on: init_generate_tasks.py
@author: Rafal Chmielewski
@team: Airflow Learning
@stakeholders: People who learns
"""
from airflow.operators.dummy_operator import DummyOperator
from airflow.operators.postgres_operator import PostgresOperator
from airflow.models import DAG, Variable, datetime
```

Then we gonna retrieve informations that init pipeline generate for us in the Variable object. GENERAL_VAR will be the same dictionary that we created in init pipeline.

```
GENERAL_VAR = Variable.get('generate_tasks', deserialize_json=True)
```

Next step i classic dag declaration with our start and end tasks.

```
dag = DAG(
    dag_id='generate_task_example',
    schedule_interval=None,
    start_date=datetime(2020, 1, 1),
    default_args={"owner": "airflow_lesson"}
)

start = DummyOperator(
    task_id='start',
    dag=dag
)

end = DummyOperator(
    task_id='end',
    dag=dag
)
```

```
task_list = []

for i in range(0, len(GENERAL_VAR['table_list'])):
    task_list.append(
        PostgresOperator(
            task_id = f'{GENERAL_VAR["table_list"][i]}_{i}',
            sql=f'SELECT count(*) from other.{GENERAL_VAR["table_list"][i]}',
            postgres_conn_id=GENERAL_VAR['psql_conn_id'],
            pool=GENERAL_VAR['pool'],
            dag=dag
        )
    )
```

And now magic of generating the parallel tasks.
In first place we need an empty list that will store all of our tasks

Then in loop we go through all list and for each element we create an new tasks of Postgres operator that will do count() on the table. In this case we use f-strings to generate sql statement for database. Adding pool from our dictionary to all tasks will make sure that only 2 of them run in the same time.
And in the end we build our task dependencies:

```
start >> task_list >> end
```

Now you may ask ok, I can generate task in parallel. Can I do so sequentially? The answer is - yes you can! And here is how:

```
"""
This pipeline shows how generate task sequentially.

Data_in: None
Data_out: None
Depend_on: None
@author: Rafal Chmielewski
@team: Airflow Learning
@stakeholders: People who learns
"""
from airflow.operators.dummy_operator import DummyOperator
from airflow.models import DAG, datetime

dag = DAG(
    dag_id='sequential_generation_example',
    schedule_interval=None,
    start_date=datetime(2020, 1, 1),
    default_args={"owner": "airflow_lesson"}
)
```

```
start = DummyOperator(
    task_id='start',
    dag=dag
)

end = DummyOperator(
    task_id='end',
    dag=dag
)

middle_task = DummyOperator(
    task_id="middle_task",
    dag=dag
)

sequential_tasks = []
for s in range(0, 6):
    sequential_task = DummyOperator(
        task_id=f"sequential_task_{s}",
        dag=dag
    )

    sequential_tasks.append(sequential_task)
    if s == 0:
        middle_task >> sequential_tasks[s]
    else:
        sequential_tasks[s - 1] >> sequential_tasks[s]
        if s == 5: sequential_tasks[s] >> end

start >> middle_task
```

Trick here is similar, but instead of putting list of task in dependency - in each iteration of loop we build the relations last object to new object. Only the first and last objects from list have to be connected to tasks outside the loop, that's why we use if statements. And because we build dependencies in loop, we don't have to put the list object into dependencies in the end of file.

xcom Object

As we said before, xcoms are objects that allows to move information between tasks using context of task instance that have added parameter in declaration provice_context=True. Let's look how to use them in practice. As always our pipeline will start with docstring and needed imports.

```
"""
This pipeline shows how use xcoms in Airflow.
Data_in:
    Postgres:
        - europe.close_relations_2015
        - europe.gdp_2016
        - europe.low_savings_2016
Data_out: None
Depend_on: None
@author: Rafal Chmielewski
@team: Airflow Learning
@stakeholders: People who learns
"""
from airflow.operators.dummy_operator import DummyOperator
from airflow.operators.postgres_operator import PostgresOperator
from airflow.operators.python_operator import PythonOperator
from airflow.operators.bash_operator import BashOperator
from airflow.models import DAG, datetime
import logging
```

Now in our examples we gonna use PythonOperator and BashOperator, because both of them are able to push information into xcom object. We gonna start with two python functions.

```
def python_xcom(**context):
    """
    generate xcom by return statement.
    :param context:
    :return:
    """
    return 'europe.close_relations_2015'


def python_xcom2(**context):
    """
    Push xcom using function
    :param context:
    :return:
    """
    ti = context['ti']
    ti.xcom_push(key='my_xcom', value='europe.gdp_2016'
```

Please look that in both function as a parameter we put kwargs (usually in pipelines named as context), because we have to be able to access tasks instance details that are saved there.

Python operator by default put everything that function returning in xcom, with key which is name reference to it as return_value.
Another approach especially when you want that PythonOperator push more than one xcom to grab task instance object from the context and then use xcom_push method to push it into task instance details.

Please keep in mind that xcom are saved as well as Variable object in backend database, so you ***cannot upload there big information like file data****. It should be just enough information for other task to eventually locate file in different source like S3 bucket.*

Next function in our code is reading xcom from context.

```
def get_xcom(**context):
    """
    retrieving xcom in python operator
    :param context:
    :return:
    """
    ti = context['ti']
    data = ti.xcom_pull(task_ids='xcom_from_bash', key='return_value')
    logging.info(data)
```

As you can see here we do similar thing as in second function. First we retrieving the task instance object from context and then using xcom_pull method we read the value of xcom that was generated by task xcom_from_bash that key is return_value.
Next part is already familiar to you - declaration of dag, start and stop tasks

```
dag = DAG(
    dag_id='xcom_example',
    schedule_interval=None,
    start_date=datetime(2020, 1, 1),
    default_args={"owner": "airflow_lesson"}
)

start = DummyOperator(
    task_id='start',
    dag=dag
)

end = DummyOperator(
    task_id='end',
    dag=dag
)
```

Next is xcom generation from BashOperator

```
ex1 = BashOperator(
    task_id='xcom_from_bash',
    bash_command="echo 'europe.low_savings_2016' ",
    xcom_push=True,
    dag=dag
)
```

By bash command parameter you can pass any command that you could run in unix terminal - in this case we want to just print out europe.low_savings_2016. By parameter xcom_push we tell airflow that according to the documentation last print to stout have to be push to xcom. It will get key return_value. the same one as our first python function.

```
ex2 = PythonOperator(
    task_id='xcom_python_return',
    python_callable=python_xcom,
    provide_context=True,
    dag=dag
)

ex3 = PythonOperator(
    task_id='xcom_python_push',
    python_callable=python_xcom2,
    provide_context=True,
    dag=dag
)
```

Next step is declare tasks for executing our python functions that generate xcoms.
Now let's also declare tasks that are using xcoms.

```
p1 = PythonOperator(
    task_id='use_xcom1',
    python_callable=get_xcom,
    provide_context=True,
    dag=dag
)

p2 = PostgresOperator(
    task_id='use_xcom2',
    sql="SELECT count(*) FROM {{ ti.xcom_pull(task_ids='xcom_python_return',
key='return_value')}}",
    postgres_conn_id='airflow_docker_db',
    dag=dag
)

p3 = PostgresOperator(
    task_id='use_xcom3',
    sql="SELECT count(*) from {{ task_instance.xcom_pull(task_ids='xcom_python_push',
key='my_xcom')}}",
    postgres_conn_id='airflow_docker_db',
    dag=dag
)
```

In use_xcom1 we get xcom from context inside python code as we explained before, but in p2 and p3 we use tempating in parameters (more about that soon)

```
{{ task_instance.xcom_pull(task_ids='xcom_python_push', key='my_xcom')}}
{{ ti.xcom_pull(task_ids='xcom_python_return', key='return_value')}}
```

Those two statement are equal to each other and {{ }} inform airflow that here value inside it have to be processed by templating engine in which we can put any valid python code to be executed and result of it put into string. task_instance and ti are reference to the same object in context, so here you can use construction that is more comfortable for you.

Last but not least is declaration of dependency which will look like this:

```
start >> ex1 >> p1 >> end
start >> ex2 >> p2 >> end
start >> ex3 >> p3 >> end
```

Templating

Airflow have built in template engine called Jinja Template[12] *which allow to pare files or string and fill out them dynamically with data that we want. Inside a template you can put any valid python code which during parsing will be executed and result of it put back in string object that is returned by engine. Also we can use clear python way of entering values into string - fstring syntax, but better to see them in examples. List of fields that and file extensions that will be template you can find in the operator code declared as variables template_fields and template_ext.*

First let's declare new dag with all needed imports, docstring and one variable.

```
"""
This pipeline shows how templating works.

Data_in: None
Data_out: None
Depend_on: None
@author: Rafal Chmielewski
@team: Airflow Learning
@stakeholders: People who learns
"""
from airflow.operators.dummy_operator import DummyOperator
from airflow.operators.postgres_operator import PostgresOperator
from airflow.operators.python_operator import PythonOperator
from airflow.models import DAG, datetime
import random

table = 'public.log'
```

Next as in last chapter - we gonna make a function that generates xcom for us.

```
def fn_generate_xcom(**context):
    """
    generate xcom by xcom_push function
    :param context:
    :return:
    """
    ti = context['ti']
    ti.xcom_push(key='my_xcom', value='2+2')
```

[12] https://jinja.palletsprojects.com/en/2.11.x/

Now let’s declare our dag

```
dag = DAG(
    dag_id='template_example',
    schedule_interval=None,
    start_date=datetime(2020, 1, 1),
    default_args={"owner": "airflow_lesson"},
    template_searchpath=['/usr/local/airflow/sql']
)
```

As you can see we have here new parameter - template_searchpath. If we want to template file in Airflow we don’t have to provide an absolute path to the file. We can just provide the relative path from one of folders that we declare in this parameter. By default path to folder where is your pipeline is already added. Jinja will search folder according to the order in list provided in parameter.
Let’s add our standard start, end and xcom generation tasks.

```
start = DummyOperator(
    task_id='start',
    dag=dag
)

end = DummyOperator(
    task_id='end',
    dag=dag
)

generate_xcom = PythonOperator(
    task_id='generate_xcom',
    python_callable=fn_generate_xcom,
    provide_context=True,
    dag=dag
)
```

First type of templating - fstring

```
ex1 = PostgresOperator(
    task_id='template_fstring',
    sql=f'SELECT count(*) FROM {table};',
    postgres_conn_id='airflow_docker_db',
    dag=dag
)
```

Here we use classic fstring from python3 that will put value of variable table that we declared at beginning of pipeline inside the string.

Second type file

```
ex2 = PostgresOperator(
    task_id='template_file',
    sql='template_file.sql',
    postgres_conn_id='airflow_docker_db',
    dag=dag
)
```

Here as you see in sql parameter we just pass the relative path to file from our sql folder. In file we can find:

```
SELECT count(*) FROM public.dag;
```

What does it mean to Jinja? It will pick up file, see that there is only text in it and pass all of this text as string to operator which will execute it on database.

Third type template in file

```
ex3 = PostgresOperator(
    task_id='template_in_file',
    sql='template_in_file.sql',
    postgres_conn_id='airflow_docker_db',
    params={"my_param": 'follow_me'},
    dag=dag
)
```

Here situation so far looks the similar as in previous example. We just added parameter params with dictionary, but in file we can find:

```
  {% set my_lovely_variable = yesterday_ds %}
SELECT
    {% for i in range(1, 5) %}
        {% if loop.index > 1 %}
         ,
        {% endif %}
        {% if i%2==0 %}
            {{i}}
        {% else %}
            {{i}} || 'is prime'
        {% endif %}
    {% endfor %}
    , 'my_lovely_variable=' || {{ my_lovely_variable }};

SELECT 1 as "{{ params.my_param }}" ;
```

Let’s go one by one here.

```
  {% set my_lovely_variable = yesterday_ds %}
```

Here we declare variable that will be visible only in this file. yesterday_ds is one of the macros provided by airflow - which are standard variables for every dag. You don't have to import them in objects that gonna be templated as they are part of Airflow implementation of Jinja engine. List of all of them you can find in Airflow documentation[13]*.*

```
 {% for i in range(1, 5) %}
      {% if loop.index > 1 %}
       ,
      {% endif %}
      {% if i%2==0 %}
          {{i}}
      {% else %}
          {{i}} || 'is prime'
      {% endif %}
   {% endfor %}
```

Here we declare loop through numbers from 1 to five and manipulate output based on condition in if statement. The equivalent in python will be:

```
string = ''
for i in range(1,5):
    add_to_string = ""
    if i > 1:
        add_to_string +=","
    if i%2 ==0:
        add_to_string+= str(i)
    else:
        add_to_string += f"{i} || 'is_prime'"
    string += f"{add_to_string} \n"
```

```
    , 'my_lovely_variable=' || {{ my_lovely_variable }}
;
```

And here we use variable declared earlier in the sql file by putting it in double {}.

```
SELECT 1 as "{{ params.my_param }}" ;
```

Last is reference to our dictionary passed to operator where my_param is a key in that dictionary. Keep in mind that this option was provided by operator itself, so if your not sure if there is such possibility, check operator and its parents documentation.

[13] https://airflow.apache.org/docs/stable/macros-ref.html

In this case logic in Airflow will be:

- *Realise that we want to pick up data for parameter sql from the file*
- *Realise that inside file there is declaration of templates.*
- *Jinja will execute all template declaration inside*
- *After rendering file - string is returned and passed to operator for next steps.*

Fourth type - xcom in parameter

```
ex4 = PostgresOperator(
    task_id='xcom_in_sql',
    sql="SELECT {{ ti.xcom_pull(key='my_xcom', task_ids='generate_xcom') }};",
    postgres_conn_id='airflow_docker_db',
    dag=dag
)
```

As you remember we already have used this construction in chapter about xcoms. For Airflow when it will parsing sql parameter it will trigger template engine to connect to backend database, get value of our xcom and pass it to string.

Fifth type - xcom in file

```
ex5 = PostgresOperator(
    task_id='xcom_in_sql_file',
    sql='xcom_in_sql_file.sql',
    postgres_conn_id='airflow_docker_db',
    dag=dag
)
```

```
SELECT {{ ti.xcom_pull(key='my_xcom', task_ids='generate_xcom') }} as field;
```

Here we have mix from second and fourth approach. the same code from third example we put in separate file. Even that Airflow triggers templating because of extension in file, then realises that is also template in file itself, so code in {} is evaluated. As we want to grab information from the xcom Airflow will retrieve it from backend database and put it into our string.

Sixth type - xcom in fstring construction

```
ex6 = PostgresOperator(
    task_id='xcom_in_fstring',
    sql=f"SELECT {{{{ ti.xcom_pull(key='my_xcom', task_ids='generate_xcom') }}}} AS
{random.choice(['x','y'])}",
    postgres_conn_id='airflow_docker_db',
    dag=dag
)
```

Here to better understand how Airflow will deal with this construction we have to look into each step of it's evaluation. This is our origin string.

```
f"SELECT {{{{ ti.xcom_pull(key='my_xcom', task_ids='generate_xcom') }}}} AS
{random.choice(['x','y'])}"
```

First python will evaluate classic fstring and return its value to the main one. Let's assume that in our case return value from random.choice() will be x. We will get this:

```
"SELECT {{ ti.xcom_pull(key='my_xcom', task_ids='generate_xcom') }} AS x"
```

As you spotted - two pairs of {} brackets are dropped. Now this string will be passed to template engine, which will recognize template in it for grabbing xcom value, so airflow will again connect to backend database, download value of this xcom and put it back into string. So in the end we gonna end up with:

```
"SELECT 2+2 AS x"
```

And this statement will be executed on the database that we declare in postgres_conn_id parameter in operator instance.

```
start >> generate_xcom >> ex1 >> ex2 >> ex3 >> ex4 >> ex5 >> ex6 >> end
```

Last but not least we have declaration of our task dependency.

BranchingOperator

Sometimes you will want to have a process that is able to only part of tasks depends on some conditions or result of previous steps. In this cases you can use branching operator which were made especially for that.

Let’s see how it goes in practice.

```
"""
This pipeline example how branching operator works.

Data_in: None
Data_out: None
Depend_on: None
@author: Rafal Chmielewski
@team: Airflow Learning
@stakeholders: People who learns
"""
from airflow.operators.dummy_operator import DummyOperator
from airflow.operators.python_operator import BranchPythonOperator
from airflow.models import DAG, datetime
from random import choice
```

First part as always will be docstring with description of pipeline and all needed imports.

```
def make_choice():
    """
    Function is choosing random path for run
    :return:
    """
    options = ['task_1', 'task2', 'task_3']
    return choice(options)
```

Next we declare Python function that will return the task_id value that will be followed after BranchingOperator.

```
dag = DAG(
    dag_id='branching_example',
    schedule_interval=None,
    start_date=datetime(2020, 1, 1),
    default_args={"owner": "airflow_lesson"}
)

start = DummyOperator(
    task_id='start',
    dag=dag
)
```

Then standard definition of dag and our start task.

```
do_choice = BranchPythonOperator(
    task_id='do_choice',
    python_callable=make_choice,
    dag=dag
)
```

Next is declaration of our BranchPythonOperator. As you see the list of arguments is the same here as in PythonOperator that we have used before.

```
t1 = DummyOperator(
    task_id='task_1',
    dag=dag
)

t2 = DummyOperator(
    task_id='task_2',
    dag=dag
)

t3 = DummyOperator(
    task_id='task_3',
    dag=dag
)

t4 = DummyOperator(
    task_id='task_4',
    dag=dag
)

t5 = DummyOperator(
    task_id='task_5',
    dag=dag
)

t6 = DummyOperator(
    task_id='task_6',
    dag=dag
)
```

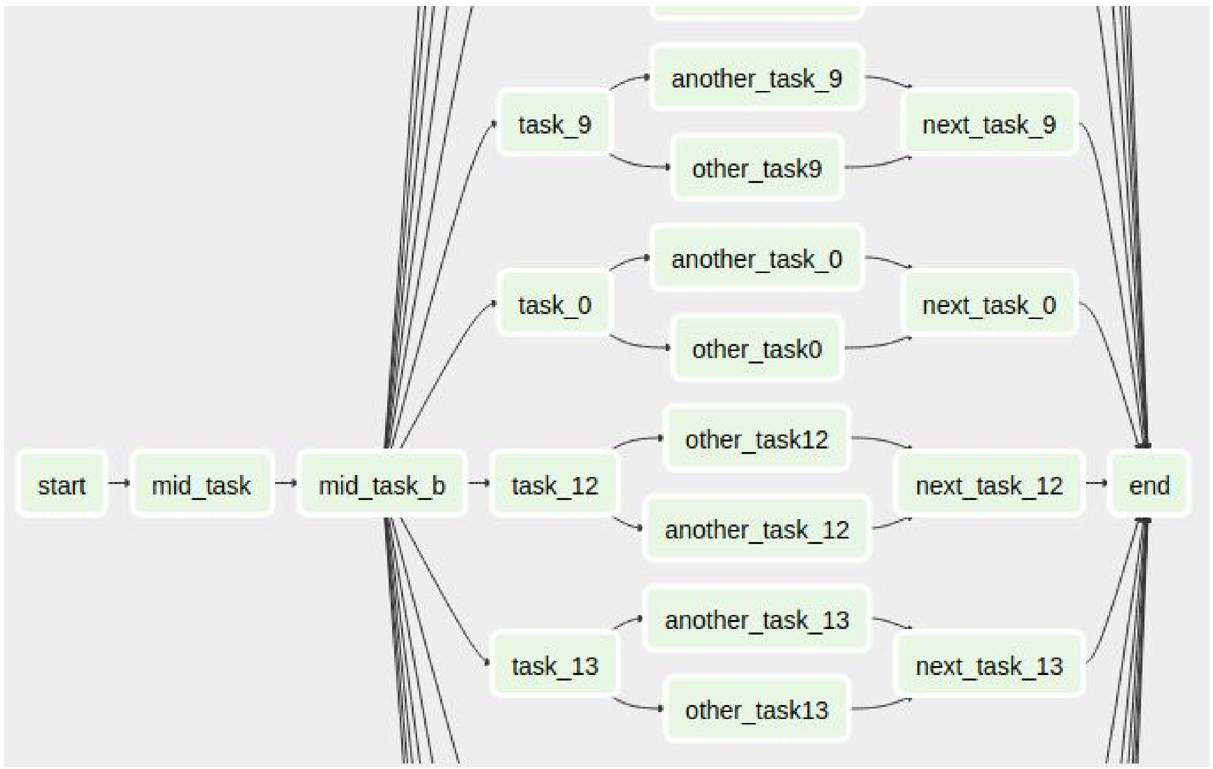
Then in the example we gonna use bunch of DummyOperators just to show three different possibilities of how process will be executed.

```
start >> do_choice
do_choice >> t1 >> t4
do_choice >> t2 >> t5
do_choice >> t3 >> t6
```

And in the end we are building our dependencies.
Very important here to remember is that BranchingPythonOperator ***must have directly downstrem dependencies with all task that can be chosen from function****.*
Another thing is by theory you can follow another step after steps in each node by using triggering rules[14]*, but in Airflow 1.10.1 that we are using this possibility have a bug that does not allow for that. It was fixed in version 1.10.3*[15]*.*

Subdag

Subdags are concept that allow to not repeat code for generating repetitive tasks and makes it easier to follow in web user interface.
Consider simple flow here:



After mid_taks_b we have many related tasks that have similar pattern. It is possible to generate this list using techniques mentioned earlier, but what if after next_task_x you will have another 2 or 3 flows with another group of tasks with x steps to execute? Graph will become very unreadable and your code will most likely repeat itself.

Let's do the same list using Subdag.

[14] https://airflow.apache.org/docs/stable/concepts.html#trigger-rules
[15] https://issues.apache.org/jira/browse/AIRFLOW-3823

```
"""
This pipeline shows how sub dags are created.

Data_in: None
Data_out: None
Depend_on: None
@author: Rafal Chmielewski
@team: Airflow Learning
@stakeholders: People who learns
"""
from airflow.operators.dummy_operator import DummyOperator
from airflow.operators.subdag_operator import SubDagOperator
from airflow.models import DAG, datetime
```

Step one - classic docstring and imports

```
def prepare_sub_dag(parent_dag, child_dag):
    """
    Preparing sub dag based on parent
    :param parent_dag:
    :param child_dag:
    :return:
    """
    sub_dag = DAG(
        dag_id=f'{parent_dag}.{child_dag}',
        start_date=datetime(2020, 1, 1),
        schedule_interval=None,
    )
    with sub_dag:
        t1 = DummyOperator(
            task_id=f'sub_dag_task1',
            dag=sub_dag,
        )
        t2 = DummyOperator(
            task_id=f'sub_dag_task2',
            dag=sub_dag,
        )
        t3 = DummyOperator(
            task_id=f'sub_dag_task3',
            dag=sub_dag,
        )
        t4 = DummyOperator(
            task_id=f'sub_dag_task4',
            dag=sub_dag,
        )
        t1 >> [t2, t3] >> t4
    return sub_dag
```

Step two - python function that will generate for us subdag

But here we need to stop for a minute and explain what and why is happening.

```
def prepare_sub_dag(parent_dag, child_dag):
```

Our function will get two arguments parent_dag and child_dag. Those two will let us to build proper naming according to conventions set by airflow developer's team.

```
    sub_dag = DAG(
        dag_id=f'{parent_dag}.{child_dag}',
        start_date=datetime(2020, 1, 1),
        schedule_interval=None,
    )
```

As you see we declare a new dag in function, it will actually be our subdug, but because we put it inside a function - airflow will not treat it as a separate dag in our file and not generate another row in dag list with it.

```
    t1 = DummyOperator(
        task_id=f'sub_dag_task1',
        dag=sub_dag,
    )
    t2 = DummyOperator(
        task_id=f'sub_dag_task2',
        dag=sub_dag,
    )
    t3 = DummyOperator(
        task_id=f'sub_dag_task3',
        dag=sub_dag,
    )
    t4 = DummyOperator(
        task_id=f'sub_dag_task4',
        dag=sub_dag,
    )
    t1 >> [t2, t3] >> t4
```

Next step is build our tasks as we did before in any other pipeline and prepare dependencies between them. Remember that those will be tasks inside your subdag.

```
    return sub_dag
```

In the end of the function we gonna just return our sub_dag created so far.

```
dag = DAG(
    dag_id='sub_dag_example',
    schedule_interval=None,
    start_date=datetime(2020, 1, 1),
    default_args={"owner": "airflow_lesson"}
)
start = DummyOperator(
    task_id='start',
    dag=dag
)

end = DummyOperator(
    task_id='end',
    dag=dag
)
```

Then as usual we declare our main dag and star/end tasks.

```
r_task = DummyOperator(
    task_id=f'some_task',
    dag=dag
)
r_task_2 = DummyOperator(
    task_id=f'another_task',
    dag=dag
)
```

To replicate schema of task from the beginning of this chapter we gonna add two standard tasks.

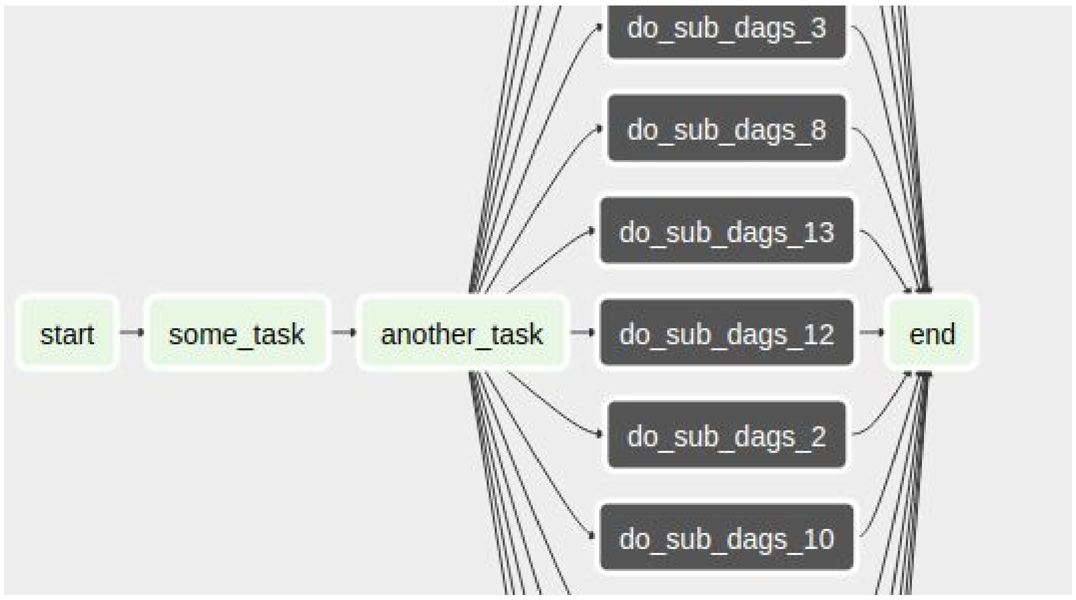
```
for i in range(0, 15):
    sub_dags = SubDagOperator(
        task_id=f'do_sub_dags_{i}',
        subdag=prepare_sub_dag(dag.dag_id, child_dag=f'do_sub_dags_{i}'),
        dag=dag
    )
    r_task_2 >> sub_dags >> end
```

Then in loop we gonna create our subdags. Please pay attention that taks_id and child_dag in subdag parameter have to be the same. Also we are allowed to already build dependency from already created tasks.

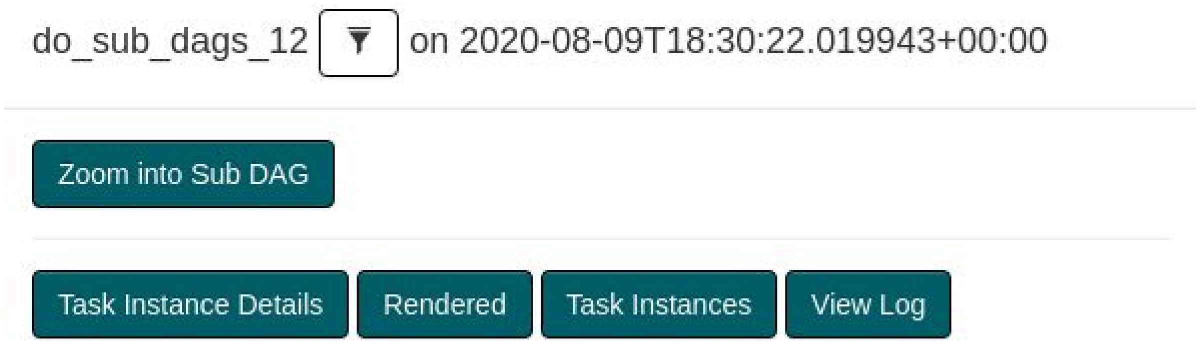
```
start >> r_task >> r_task_2
```

Last step is just add dependency for rest tasks.

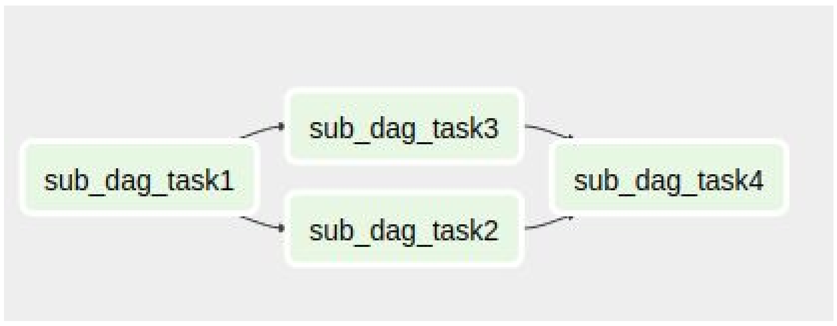
And now our pipeline looks like that:



You have to agree that is better to follow it right now!
If you want to see what is going on inside each subdag - you have to just click on it.


do_sub_dags_12 on 2020-08-09T18:30:22.019943+00:00

And at top of the window you gonna see new button - Zoom into Sub DAG which after click



Will show you a new grah will steps declared inside our subdag!

Working with queues

As was mentioned on the beginning of the document - workers are assigned to the queues and they are very specific future that can be used only with CeleryExecutor that we have in our environment. You maybe want to ask - how it could be useful? The answer is: when you need to do some task for specific worker or worker group because of compute resource or security/compliance reasons.
In this repository we have two queues: default and secret, but in environment that you will build or work if provided to you by other team they can be different. If second case when you don't know what queues are in place or you need to create one for some reason - it will be a good idea to ask your cluster administrator for an advice.

Never less let's see how to assign task to the queue.

```
"""
This pipeline shows how work with queues.

Data_out: None
Depend_on: None
@author: Rafal Chmielewski
@team: Airflow Learning
@stakeholders: People who learns
"""

from airflow.operators.dummy_operator import DummyOperator
from airflow.models import DAG, datetime


dag = DAG(
    dag_id='queues_example',
    schedule_interval=None,
    start_date=datetime(2020, 1, 1),
    default_args={"owner": "airflow_lesson"}
)

start = DummyOperator(
    task_id='start',
    dag=dag
)

end = DummyOperator(
    task_id='end',
    dag=dag
)
```

As always let's start with docstring, imports, dag declaration and start/end tasks.

```
t1 = DummyOperator(
    task_id='default_queue',
    queue='default',
    dag=dag
)
```

In first task we gonna use parameter queue and assign task to queue named default (which is a standard name of the queue when not defined other way in cluster configuration).

```
t2 = DummyOperator(
    task_id='default_queue2',
    dag=dag
)
```

The same effect we gonna have when not providing this argument at all.

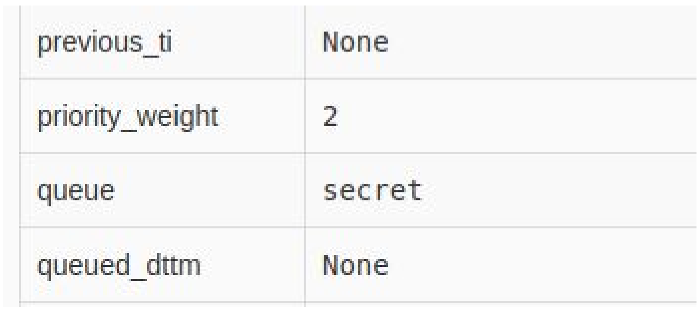
```
t3 = DummyOperator(
    task_id='secret_queue',
    queue='secret',
    dag=dag
)
```

And to pass our tasks to secret worker we just provide its name in queue parameter. There is no more magic with it!

```
start >> [t1, t2, t3] >> end
```

In the end of course our dependencies for declared tasks.

If you want to see to which queue task was sent in web user interface you can click on task and then Task Instance Details. On the list you will find:



previous_ti	None
priority_weight	2
queue	secret
queued_dttm	None

Homework

Create an dynamic DAG with init that will:

- *Query the postgres schema europe and generate Variable with all tables names*
- *Based on it main pipeline will in parallel of 2 count task at the same time will check how many records is in each table and push this information to table other.europe_tables_count. You have to create this table first.*
- *If day of the number of rows is even log all rows from table other.europe_tables_count to log in other case just end pipeline.*

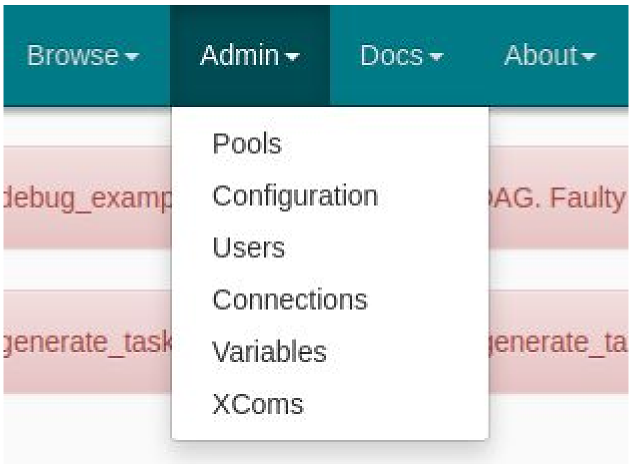
Create pipeline that will

- *Query table in Postgres public.log and retrieve max value of dttm collum. Pass this information to xcom*
- *Another task should pick up this date from point 1 and query table public.xcom to check if there is any records (by timestamp column) that came after. Log this information into xcom.*
- *Using python operator log information about dag execution date and both informations from previous two points.*

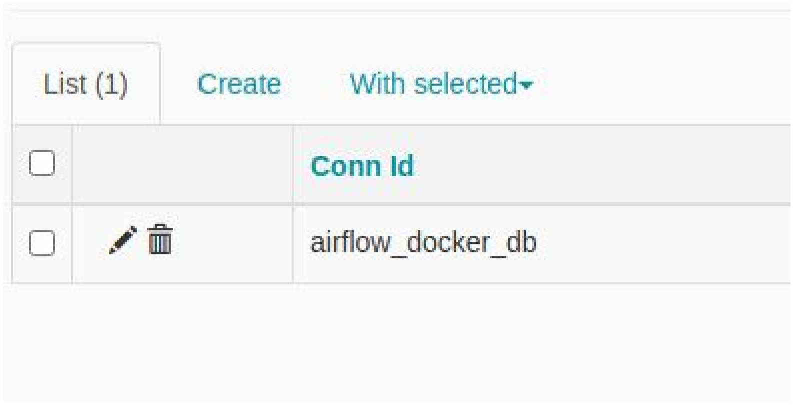
Let’s make it custom

Add connection to Airflow

When you want to interact with any external service you have to add new connection into airflow. The easiest way to do so is adding it with web user interface.

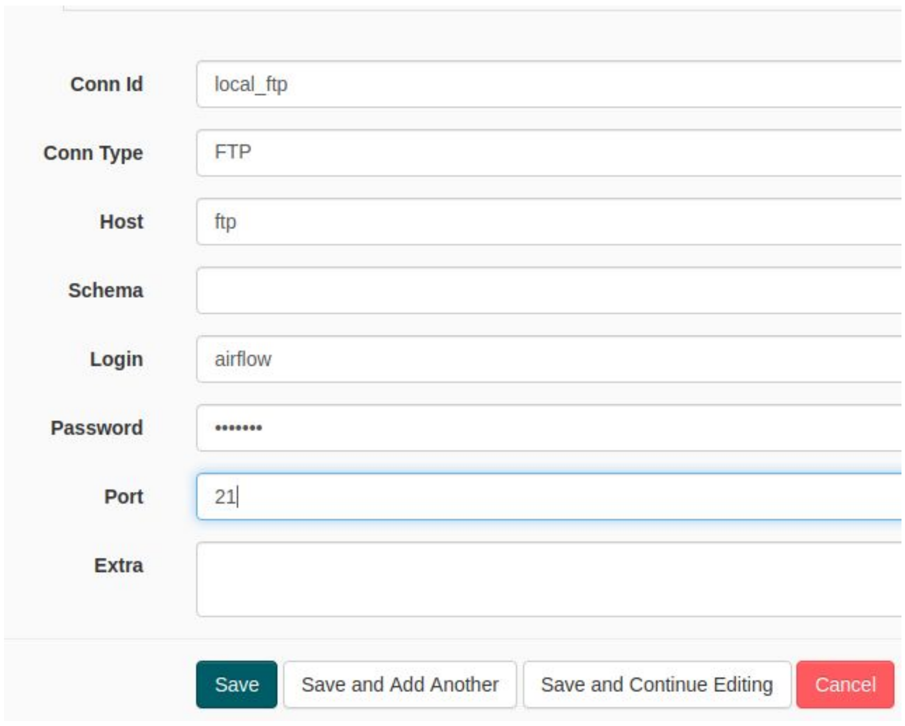


Open web interface by accessing http://localhost:8080 and from top menu choose Admin → Connections.



		Conn Id
☐	✎ 🗑	airflow_docker_db

You will see currently accessible connections in cluster. Let’s add new one by clicking Create.



Conn Id	local_ftp
Conn Type	FTP
Host	ftp
Schema	
Login	airflow
Password	•••••••
Port	21
Extra	

Now we have to fill out a form with all needed information.
Some hooks can also use additional configuration from Extra field that have to be filled in json format. In our case we will add connection to ftp service included in our environment as docker container.
After you do it just click Save.

There is also possibility to add connection using Airflow CLI[16] by command but for that you need to have in configured environment variables AIRFLOW__CORE__SQL_ALCHEMY_CONN and FERNET_KEY with date of cluster where connection should be added. Also with Airflow CLI you cannot so far change existing connection. You have to delete old one and create it one more time with new credentials.

```
airflow connections -a --conn_id NAME --conn_uri
TYPE://USER:PASSWORD@HOST:PORT/
--conn_extra='{"extra_key1":"extra_value1","extra_key2":"extra_value_2"}'
```

Important
You should not add manually connection to any production cluster. It is ok to do it local setup as you want to test something. In work when you will want to add connection to production cluster in first place ask cluster administrator how to do it!

[16] https://airflow.apache.org/docs/stable/cli-ref

Creating Custom Plugin

Folder structure

As you know Airflow as Framework allows us to create custom plugins which are responsible for encapsulation logic of interacting with external services or data that can be reused in many workflows. Always when you create some interactions in your pipeline using for example PythonOperator you should ask yourself - Is it possible to use this functionality later on by someone else? Or it is so individual logic that only one process can use it? If you answered yes on first question - you should put this code into plugin if not put it as a function for python operator.

All plugins are store in folder that is configured in cluster. In our case it is *plugin* folder in repository.
Good practice also is have a structure of your plugins like this:

```
└──── <plugin_name>_plugin
        ├──── __init__.py
        ├──── hooks
        │       ├──── __init__.py
        │       └──── <hook_name>_hook.py
        ├──── operators
        │       ├──── __init__.py
        │       └──── <operator_name>_operator.py
        └──── sensors
                ├──── __init__.py
                └──── <sensor_name>_sensor.py
```

All files names should be written in snake case and each folder in plugin should contain *__init__.py* file. In the moment we gonna see what should be inside each of it.

As an example in this document we gonna create plugin that will allow to grab values of select statement made on postgres database and sent result as csv or parquet file into ftp service. Our plugin will have name postgres_extended_plugin.

Create custom hook

As we mentioned in the beggining of this document hooks are classes that are responsible for interaction with external data sources/services. Remember that each hook is able to interact just with one data source!

After creating the folder structure for our hook let’s work on our postgres_extended_hook.py

```
from airflow.hooks.postgres_hook import PostgresHook
from airflow.exceptions import AirflowException
import logging
import pandas as pd
```

Our file will start with all needed imports. As you see here we don’t put docstring on the beginning of the file. Also as postgres hook is already exising we gonna use it as a base for our extension.

```
class PostgresExtendedHook(PostgresHook):
```

Next step is declare our class for our hook and provide as parent already existing hook.
In case when is service that does not have any hook so far in Airflow you have to use here *BaseHook* for services that are not databases - you can import it by:
from airflow.hooks.base_hook import BaseHook
or DbApiHook for databases: *from airflow.hooks.dbapi_hook import DbApiHook*

```
    conn_name_attr = 'postgres_conn_id'
    default_conn_name = 'postgres_default'
    supports_autocommit = True

    def __init__(self, *args, **kwargs):
        super(PostgresExtendedHook, self).__init__(*args, **kwargs)
        if 'conn_name_attr' in kwargs and kwargs['conn_name_attr'] in kwargs:
            self.postgres_conn_id = kwargs[kwargs['conn_name_attr']]
            logging.debug(f"+++ Override postgres_conn_id with {self.postgres_conn_id}")
```

Then we will overwrite setting from DbApiHook and pass all provided parameters also to parent.

```
    def get_pandas_df(self, sql, parameters=None, chunk_size=None):
        """
        Method is returning pandas DF based on provided SQL query
        :param sql: str -> sql file or path to file
        :param parameters: dict -> paramters that will be used to template sql query
        :param chunk_size: how many records (maximum) each df should posses in return.
        :return:
        """
        logging.info(f"getting postgres engine: {self.get_sqlalchemy_engine()} with uri:
{self.get_uri()}")

        try:
            df = pd.read_sql(sql, self.get_sqlalchemy_engine(), chunksize=chunk_size)
        except Exception as ex:
            raise AirflowException(f"Get pandas df from sql statement: {sql} has failed with
exception : {ex}")
        return df
```

And in the end our function that we want to use in postgres_external_hook.
But here we need to make some words of explanation.
As we use postgres_hook as a parent we have access to all methods that are already declared there. Also get_pandas_df is there but here we gonna overwrite it with our own logic. As you see in our case we let user to use pandas parameter *chunksize* from method *read_slq* and we also using self.get_sqlalchemy_engine() method that is coming from DbApiHook. That is why very important to check before writing any line of code what parents of our class are already providing, so we write what need to be added or changed.
Also remember that all functions in hoos needs to have its own docstring that will documente logic of it and if you perform any work on data source/service put it in try-catch block. We need to be sure that if something goes wrong like for example timeout in connection we gonna break the execution of hook and send this information to logs. both of two actions you will achieve by using AirflowException!

Last step is just create *__init__.py* file in folder of out hook. It will be empty, but it is needed for Airflow to recognize that here is code that should be registered for using in pipelines.

That will conclude our extended hook, let’s move to the operator.

Custom operator

Operators are one of the building blocks for pipelines. Basically operators are encapsulated logic of moving data from one place to another or manipulating data on some resources in strict way. Operators are used later as task in our pipeline so process that is closed in them must be written in such way that other users or you can reuse them in another pipeline

For example in this document we gonna create operator that is dumping result of SQL query on postgres database and save it as a csv file in ftp service.

```
from airflow.operators import BaseOperator
from airflow.exceptions import AirflowException
from postgres_extended_plugin.hooks.postgres_extended_hook import
PostgresExtendedHook
from airflow.contrib.hooks.ftp_hook import FTPHook
import logging
from tempfile import NamedTemporaryFile
```

In first place we need imports for our operator.
Take a closer look to it. We gonna use there a NamedTemporaryFile - and reason for that is you cannot be sure that on task restart or in another step on the pipeline file will be accessible on different worker. To avoid that files after making any operation on them should be eventually saved in external storage like AWS S3 or sftp.
Also when you just read data - in case of not using NamedTemporaryFile will not be removed from the worker and in future it can lead to not having enough disc space on worker nodes. When you need to do anything on file always use NamedTemporaryFile[17], if you will need to make some work on few of them - then use TemporaryDirectory[18].

In our case we don't have any operator that could be our parent, so we gonna use a Airflow default one BaseOperator and declare our class PostgresFtpOperator with proper docstring:

```
class PostgresFtpOperator(BaseOperator):
    """
    Operator is responsible for dumping data from postgres into file on sftp server.

    :param postgres_conn_id -> str - connection id for Postgres DB in Airflow
    :param sftp_conn_id -> str - connection id for SFTP in Airflow
    :param sql -> str - sql query or path to file with it
    :param file_desc -> dict - Python dictionary that include name and format of file ex.
            file_desc = {
            "name": "my_file_name",
            "format" "csv" or "parquet"
            }
    """
```

[17] https://docs.python.org/3.6/library/tempfile.html#tempfile.NamedTemporaryFile
[18] https://docs.python.org/3.6/library/tempfile.html#tempfile.TemporaryDirectory

Then we need to decide which fields and file extensions in our Operator should be template by overwriting:

```
    template_fields = ('sql',)
    template_ext = ('.sql', )
    ui_color = '#ffad33'
```

In our case templated will be sql parameter and .sql files if provided in one of templated fields. Also as you can see we have here ui_color in hex syntax. It will be a background colour of our task in pipeline graph.

```
    def __init__(self,
                 postgres_conn_id: str,
                 sftp_conn_id: str,
                 sql: str,
                 file_desc: dict,
                 *args,
                 **kwargs):
        self.sql = sql
        self.postgres_conn_id = postgres_conn_id
        self.sftp_conn_id = sftp_conn_id
        self.file_desc = file_desc
        super().__init__(*args, **kwargs)
```

Next step i define a init for our class. Any information puted as a argument to __init__ function gonna be provided by end user of our pipeline. A good thing to do i also put here what types we are expecting to be passed. Also you need to assign them as a class attribute with self. On the moment of this assignment the templating is happening!
The last step is also pass any needed information to parent class.

```
    def validate_file_desc(self):
        """
        Function is validating if file_desc dictionary contain required data. If not it raise
AirflowException
        :return: dict
        """
        if 'name' not in self.file_desc.keys() or 'format' not in self.file_desc.keys():
            raise AirflowException('file_desc does not have required keys: name, format')
        elif self.file_desc['format'].lower() not in ['csv', 'parquet']:
            raise AirflowException('file_desc have incorrect format type: csv, parquet')
        else:
            return {"name": self.file_desc['name'], "format": self.file_desc['format']}
```

Another good thing to do is when you expect to get data in specific format is to check if all needed informations are in place. For that we gonna declare new method in our class that will check if file_desc parameter have all needed keys and if they have not - raise an exception so task will just fail with proper information in the logs. Every method declared for class also should have docstring with description what is happening inside.

```
    def execute(self, context):
        """
        Main execution point. Steps that are done
            1) connecting to Postgres DB
            2) queering DB and pull data into pandas dataframe
            3) dump data from dataframe into file
            4) send file on SFTP server.
        :param context:
        :return: none
        """
        logging.info(f'Preparing dataframe...')
        psql_hook = PostgresExtendedHook().get_hook(self.postgres_conn_id)
        df = psql_hook.get_pandas_df(sql=self.sql)
        logging.info('Writing data into temp file')
        with NamedTemporaryFile() as f:
            if self.file_desc['format'].lower() == 'csv':
                df.to_csv(f.name, sep=';', quotechar='|')
            if self.file_desc['format'].lower() == 'parquet':
                df.to_parquet(f.name, engine='pyarrow')
            try:
                logging.info('Sending file to FTP')
                print(self.sftp_conn_id)
                ftp_hook = FTPHook(ftp_conn_id=self.sftp_conn_id)
                conn = ftp_hook.get_conn()
                f.flush()
                conn.set_pasv(False)
                conn.storbinary(f'STOR {self.file_desc["name"]}.{self.file_desc["format"]}',
open(f.name, 'rb'), 1)

            except Exception as ex:
                raise AirflowException(f'Could not put file on SFTP. Details {ex}')
```

Last part is execute method which is required to implement in every operator as Airflow during execution of pipeline will execute this one. Here should be written all logic behind the operator, but if its to long or you want to split problem into smaller parts - just create as many additional methods as needed. In this case we don’t have that much code it can stay in one place.

As you can see here we are using two hooks - one that we created earlier PostgresExtendedHook and FtpHook that was provided by Airflow to grab data from one point and push it to another. This operation can be done by other pipelines, that's why it goes to custom operator and not for python code to use in PythonOperator.

Also here you should put docstring with information what is happening inside the execute method so any other person can quickly understand if operator can be used for some process or not.

Custom Sensor

Sensors are operators that have very specific purpose. Those are used to detect any changes on external services/data sources that are needed for pipeline to work on. This is especially important when process that you don’t know is also doing some work on data source. For example pushing file to ftp that should be picked up by airflow or uploading some data into database.

Here we gonna do a simply sensor that will check for us if some of table is existing in database.

The beginning of the file is nearly the same as for operator:

```
class PostgresTableSensor(BaseSensorOperator):
    """
    Sensor is checking if table is able to query and let dag run further if that's possible.
    :param postgres_conn_id: str -> name of Airflow connection ID
    :param table: str -> name of table in database
    :param schema: str -> name of schema in database
    """

    @apply_defaults
    def __init__(self,
                 postgres_conn_id: str,
                 table: str,
                 schema: str,
                 *args,
                 **kwargs):
        self.postgres_conn_id = postgres_conn_id
        self.table = table
        self.schema = schema
        super().__init__(*args, **kwargs)
```

Here also you can define template fields and extensions, but in our case they are not needed so we will just skip them and leave to inherited from parents.

```
    def poke(self, context):
        """
        Main execution point. Function is trying to run select query against table. If that was
possible
        return true
        :param context:
        :return: bool
        """

        try:
            logging.info(f'Trying to query table {self.schema}.{self.table}')
            psql_hook = PostgresHook(self.postgres_conn_id)
            _ = psql_hook.run(f"SELECT * FROM {self.schema}.{self.table} LIMIT 1;")
        except Exception as ex:
            logging.info(f'Cannot query table. Details: {ex}')
            return False

        return True
```

The biggest difference is method that Airflow will call during execution for sensors is poke - so it always should be implemented.
that function depending on your conditions have to return bool value of True - when we want to let airflow to execute next tasks from the pipeline or False when we want to wait and if timeout parameter for task will be hitted - fail the whole process.
In this case we gonna try to run select statement against database and see if any error form it was received.

Put it together into plugin

After creating all blocks of our plugin we need to register it, so Airflow will recognize the imports used in pipelines and let use our custom code there. To do so we have to create __init__.py file in our main plugin catalog that will look like this:

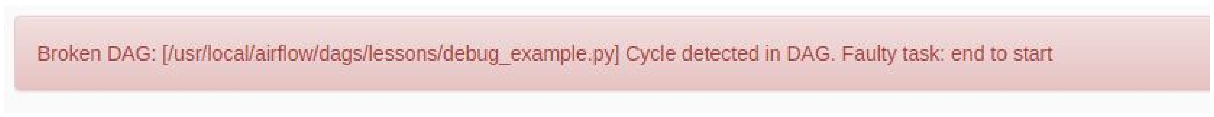
```
from airflow.plugins_manager import AirflowPlugin
from postgres_extended_plugin.hooks.postgres_extended_hook import
PostgresExtendedHook
from postgres_extended_plugin.operators.postgres_ftp_operator import
PostgresFtpOperator
from postgres_extended_plugin.sensors.postgres_table_sensor import
PostgresTableSensor


class PostgresExtendedPlugin(AirflowPlugin):
    name = "postgres_extended_plugin"
    hooks = [PostgresExtendedHook]
    operators = [PostgresFtpOperator]
    sensors = [PostgresTableSensor]
```

So first part is import of all our custom code and AirflowPlugin from the plugin manager. Then we have to extend the class of Airflow plugin with our own in camel case and provide there list off all custom classes that we build.
Well done! We made our first plugin. You can see it in accion in dags psql_ftp_example and sensor_example, but at this moment you should be able to write those dags by yourself.

Debug - where to find logs

There are several places where you can find logs for working with Airflow.

1. Task log in web user interface → Can be accessible when you click on task in your pipeline and then Logs. So you gonna find there any traceback from operators, hooks and sensors.
2. Red block from web user interface.

   ```
   Broken DAG: [/usr/local/airflow/dags/lessons/debug_example.py] Cycle detected in DAG. Faulty task: end to start
   ```

   Logs and traceback from them are visible only in standard output of web server process. In case of our docker environment you can use *docker container logs teaching_airflow_webserver_1* command in your terminal to find traceback for the issue.
3. Backend Database public.logs
   Here you can find information where task was executed and what command was used for it. Most likely it will be used only by cluster administrator

Homework:

- Add connection to local MySql database
- Create a plugin (Operator and needed hooks) that will:
  1. Query Mysql database public.avocado and pack data into JSON
  2. Write JSON data into Postgres other.avocado table. It's not existing so create it!
  3. If number of rows is even - store file on SFTP (CSV format)
- Write a sensor that will check if the file from first task is on ftp. If yes log the first five lines of it. Sensor should run in parallel to file generation tasks in pipeline.